C++ Course 11 : Templates.

2020 by Oleksiy Grechnyev

Comparison function

Compare two **int** numbers (returns 0, +-1) :

```
int compareInt(int a, int b) {
        if (a < b)   return -1;
        else if (b < a)  return 1;
        else  return 0;
}
```

Same for **double** numbers:

```
int compareDouble(double a, double b) {
        if (a < b)   return -1;
        else if (b < a)  return 1;
        else  return 0;
}
```

Same for **unsigned long long, float, short, char, string** ...

Lots and lots of types !

Comparison function (overloaded)

Overloading : Use the same name **compare()** for all types.

Compare two **int** numbers (returns 0, +-1) :

```
int compare(int a, int b) {
        if (a < b)   return -1;
        else if (b < a)  return 1;
        else  return 0;
}
```

Same for **double** numbers:

```
int compare(double a, double b) {
        if (a < b)   return -1;
        else if (b < a)  return 1;
        else  return 0;
}
```

Same for **unsigned long long, float, short, char, string** ...

Lots and lots of types ! What about mixed types? **int compare(double a, float b)**

Solution : Templates

```
template <typename T>
int compare(const T & a, const T & b) {
        if (a < b)
                return -1;
        else if (b < a)
                return 1;
        else
                return 0;
}
...
compare(3, 4)                   // int, automatic type deduction
compare(3, 2.5)                 // ERROR !!!
compare<double>(3, 2.5)         // double, explicit type (int converted to double)
compare(string("ABC"), "AB"s)   // string, automatic type deduction
compare<string>("ABC", "ABCD")  // string, explicit type (const char * converted to string)
```

What happens if we use a wrong type?

```
template <typename T>
int compare(const T & a, const T & b) {
        if (a < b)
                return -1;
        else if (b < a)
                return 1;
        else
                return 0;
}
...
compare(3, 4)                    // T == int, this is OK because int has operator<
```

Now let us try:

compare(cout, cerr)

Instantiation: **T == ofstream**. Trying to compile **compare<ofstream>()** .

Compile Error ! No operator< !

Containers

ContainerInt ic(3);
ContainerString sc{"John", "Jim", "Jane"};
Lots and lots of types ! Solution : *class templates* :

```
template <typename T>
class Container{
...
private:
      T * data;
};
```

All C++ containers are templates.
Most other standard library classes are templates:
basic_string, basic_istream, shared_ptr, function, future ...

These are synonyms (**class** was used before C++ 11) :
<typename T> and **<class T>**

How do C++ templates work?

- Every time we use **compare()** in our code, it's *instantiated*
- *Instantiation* means replacing **T** with a concrete type, e.g. **compare<int>()**
- Compiler generates different binary code for **compare<int>()** and **compare<char>()**
- Instantiation takes place at *compile time*
- C++ templates use *duck typing* (If it quacks like a duck ...)
- Any type **T** which has **operator<** will be compiled successfully
- No need for any common parent class/ interface (not OOP style !)
- Any type **T** which has NO **operator<** will give compiler error
- Library templates are always in .h files, never in .cpp (like class definitions) !
- This includes functions and definition of all methods! All inline!
- Local templates in a .cpp file can only be used in the same file
- Library templates are in *.h files, not in .o/.a or .so/.dll, never pre-compiled !
- We cannot pre-compile **compare()** before we know argument type !

Once again, how does it all work?

1. The compiler reads the template definition. No code is generated. Minimal syntax checks.

```
template <typename T>
int compare(const T & a, const T & b) { ... }
```

2. The compiler sees :

```
compare(3, 4)
```

3. (*Type deduction*) The type **T** is *deduced* to be **int**.

4. (*Instantiation*) The instance **compare<int>()** is generated and compiled. Compile errors can happen at this stage!

5. The code to call **compare<int>()** is generated

6. The code for **compare<int>()** is reused if used more than once (at least in the given .cpp file).

Can we compare two different types?

```
template <typename T, typename U>
int compare(const T & a, const U & b) {
        if (a < b)
                return -1;
        else if (b < a)
                return 1;
        else
                return 0;
}
...
compare(3, 4)                   // int, int
compare(3u, 4ull)               // unsigned int, unsigned long long
compare(3, 2.5)                 // int, double
compare(3.0f, 2.5)              // float, double
```

Now different types can be compared! But beware of mixing signed and unsigned types!

Three (or four) language tiers of C++

4. Bonus tier: CMake and other build systems, if used.
CMake (make, environment) variables live here.

3. Preprocessor: processes **#include** , **#ifdef** , **#define** etc.

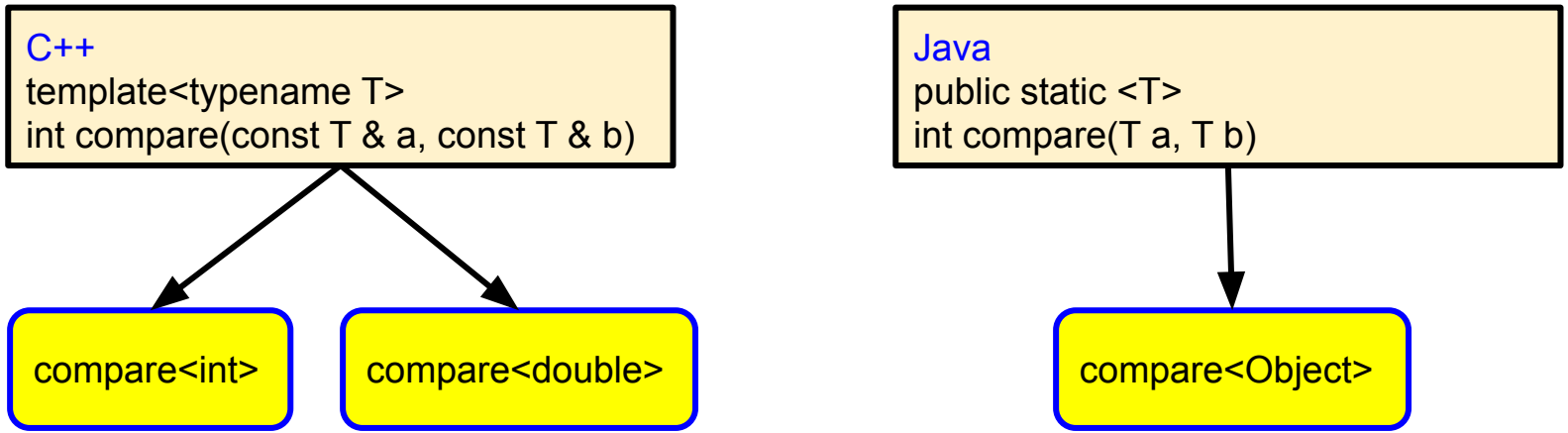
2. Compile time: Templates, **auto** , **decltype()**, **constexpr**.
Template metaprogramming is a Turing-complete language!
Templates are "executed" at the compile time.

1. Actual code compilation. Machine code is generated.

C++ templates vs Java Generics

1. C++ templates are instantiated at *compile time*. Separate binary code is generated for each instance ! Based on *duck typing* !

2. Java Generics are instantiated at *run time*. A common binary code is generated for a common parent, usually **Object**. Based on *class polymorphism* !

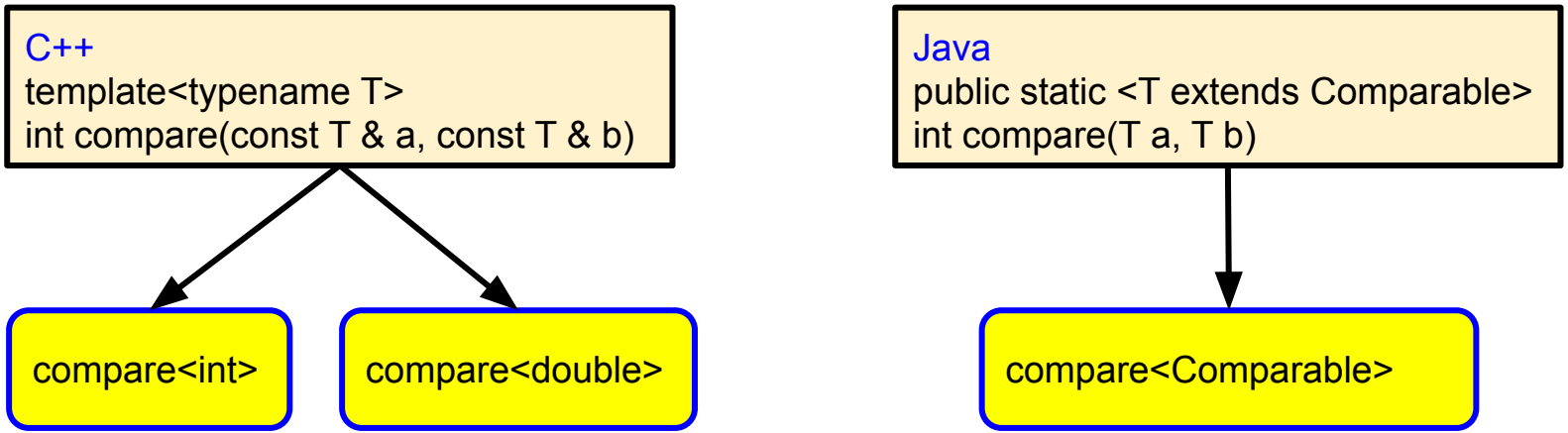


But Java class **Object** cannot be compared with **<** !!!

C++ templates vs Java Generics (Comparable interface)

1. C++ templates are instantiated at *compile time*. Separate binary code is generated for each instance ! Based on *duck typing* !

2. Java Generics are instantiated at *run time*. A common binary code is generated for a common parent, usually **Object**, but here **Comparable** (an interface). Based on *class polymorphism* !



With interface **Comparable** everything should work. Extra slides : C++ vs Java table.

Template specialization

A template function for type **T** :

```
template <typename T>
int compare(const T & a, const T & b) {   // For all T != float
    if (a < b) return -1;
    else if (b < a) return 1;
    else return 0;
}
```

Suppose we want a different behavior for type **float** only :

```
template <>
int compare(const float & a, const float & b) { // For T == float only !
    if (a < b) return -7;
    else if (b < a) return 7;
    else return 0;
}
```

Note : this is similar to an overload, but different in details.

int compare(const float & a, const float & b) {...}

Non-type arguments: Multiply by N

Multiply something by a compile-time **int** number.

```
template <int N, typename T>  // N is first, T can be deduced
T mul(const T & t) {
    return t*N;
}
```

N is an **int** argument (compile-time number), cannot be deduced!
N is a regular C++ type argument (i.e. not **typename**/**class**).
N must be a literal or **constexpr** (available at compile time).

```
cout << "mul<3>(2.1) = " << mul<3>(2.1) << endl;                  // double, deduced
cout << "mul<3, float>(2.1) = " << mul<3, float>(2.1) << endl;    // float

int i = 17;
cout << mul<i, int>(2) << endl;              // Error, i is not constexpr !
```

Read it yourself : Template default arguments

Container operations : range for

Print a container (or even built-in array) using range **for** :

```
template <typename C>
void print1(const C & c){
    for (const auto & e : c)
        cout << e << " ";
    cout << endl;
}
```

Can we do the same without **auto** (does not work with built-in arrays) :

```
template <typename C>
void print2(const C & c){
    for (const typename C::value_type & e : c)
        cout << e << " ";
    cout << endl;
}
```

What on Earth is **typename** ??? Does not work without it!

typename keyword, the real reason for it

Suppose **C** is a type parameter of a template.
We know absolutely nothing of it before instantiation (**int** ?, **vector<string>** ?, **ofstream** ?).
What is **C::value_type** ? There are 2 options:

1. It can be a *type* defined in the class **C**.
C::value_type * b; // Pointer definition

2. It can be a *static member* of the class **C**.
C::value_type * b; // Multiplication

The compiler always assumes option 2 (static member) by default.

Use **typename** (NOT **class**) for option 1:
typename C::value_type * b;

Secondary use: **typename** as a synonym to **class** in template arguments (C++ >= 11).

Container operations : iterators

Print a container (or array) using iterators :

```
template <typename C>
void print3(const C & c){
    for (auto it = cbegin(c); it != cend(c); ++it)
        cout << *it << " ";
    cout << endl;
}
```

Possible implementation of **std::find()** :

```
template <typename I, typename T>
I myFind(const I & first, const I & last, const T & val){
    for (I it = first; it != last; ++it)
        if (val == *it)
            return it;
    return end;
}
```

In this example we return the known type **I** (same as the type of **first**, **last**).

But what if we don't know the return type ?

Find first negative number in a container

Stupid version : specify the return type **T** by hand:

cout << findNeg<int>(vi.cbegin(), vi.cend()) << endl;

```
template <typename R, typename I>
const R & findNeg1(const I & begin, const I & end){
    for (I it = begin; it != end; ++it)
    if (*it < 0)
        return *it;
    throw runtime_error("NOT FOUND !");
}
```

Clever version : use **decltype** and the arrow (**->**) return type syntax:

```
template <typename I>
auto findNeg1(const I & begin, const I & end) -> const decltype(*end) & {
    for (I it = begin; it != end; ++it)
    if (*it < 0)
        return *it;
    throw runtime_error("NOT FOUND !");
}
```

Templates and built-in arrays

Size of a built-in array (NOT pointer !) compile time :

```
template <typename T, size_t N>
constexpr size_t arraySize( T (&a)[N]){ // "Ref to array" type is used here !
    return N;
}
```

Possible implementation of **begin()**, **end()** for an array (iterator is a pointer) :

```
template <typename T, size_t N>
T * myBegin(T (&a)[N]){    // "Ref to array" type is used here !
    return & a[0];
}

template <typename T, size_t N>
T * myEnd(T (&a)[N]){
    return & a[N];
}
```

Class templates: Let us try to reproduce std::array !

MyArray : my own implementation of **std::array** (more or less) :

```
template <typename T, size_t N>
class MyArray{
..
public:
    T dat[N];
}
```

MyArray : is a thin wrapper to a fixed-size built-in array.
It can be copied and moved with auto-generated constructors and assignments.
I has no explicit constructors.
The data is in the class field **dat** (not separately on heap like in **std::vector** !)
dat is public, so we can do the list initialization :

```
MyArray<string, 6> names{"Maria", "Nel", "Sophia", "Mirage", "Peppita",
"Clair"};
```

Defining types

MyArray, like standard C++ containers, defines a number of types, which are based on **T**

```
template <typename T, size_t N>
class MyArray{
public: //=== Type definitions
    using value_type = T;
    using size_type = std::size_t;
    using difference_type = std::ptrdiff_t;
    using reference = T &;
    using const_reference = const T &;
    using pointer = T *;
    using const_pointer = const T *;
    using iterator = T *;
    using const_iterator = const T *;
    ...
}
```

operator[], at()

```
template <typename T, size_t N>
class MyArray{
    ...
public:      //===== Methods
    T & operator[] (std::size_t i){ return dat[i];}  // Non-const
    const T & operator[] (std::size_t i) const { return dat[i];} // const
    T & at(std::size_t i){   // Non-const
        checkRange(i);
        return dat[i];
    }
    const T & at(std::size_t i) const {  // const
        checkRange(i);
        return dat[i];
    }
private:      //===== Methods
    void checkRange(std::size_t i) {
        if (i >= N) throw std::out_of_range("MyArray");
    }
    ...
}
```

friends, non-member operator==, one-to-one friendship

```
template <typename T, size_t N>  // Forward declaration
class MyArray;
template <typename T1, size_t N1>
bool operator==(const MyArray<T1, N1> & lhs, const MyArray<T1, N1> & rhs);

template <typename T, size_t N>
class MyArray{
    friend bool operator==<T, N>(const MyArray<T, N> & lhs,
     const MyArray<T, N> & rhs);
}

template <typename T1, size_t N1>
bool operator==(const MyArray<T1, N1> & lhs, const MyArray<T1, N1> & rhs) {
    for (size_t i = 0; i < N1; ++i)
        if (lhs.dat[i] != rhs.dat[i])
            return false;
    return true;
}
```

Template method in a template class

assign() : assign the array from a pair of iterators (NOT in **std::array**)
It's a template method of template class with its own template parameter **I**
It is defined outside class

```
template <typename T, size_t N>
class MyArray{
public:      //===== Methods
     template <typename I>
     void assign(I begin, I end);
}

template<typename T, size_t N>
template<typename I>
void MyArray<T, N>::assign(I begin, I end)
{
     int i = 0;
     for (I it = begin; it != end; ++it, ++i)
          at(i) = *it;
}
```

Iterators

```
template <typename T, size_t N>
class MyArray{
    ...
public:      //===== Iterators are pointers
    T* begin(){  return dat;}

    const T* begin() const {  return dat;}

    const T* cbegin() const { return dat;}

    T* end(){ return & dat[N];}

    const T* end() const { return & dat[N];}

    const T* cend() const { return & dat[N];}
    ...
}
```

Class template specialization example

Class template specializations:

Example 1: **std::vector<bool>** might be implemented via bitset (1 bit/bool).

Example 2: a possible implementation of **std::remove_reference** :

Usage: **remove_reference(U)::type** for any type **U** (From Lippman, C++ Primer)

```
// original, most general template (used for non-references)
template <class T> struct remove_reference {
        typedef T type;
};
// partial specializations that will be used for lvalue and rvalue references
template <class T> struct remove_reference <T&>  // lvalue references
{ typedef T type; };
template <class T> struct remove_reference <T&&> // rvalue references
{ typedef T type; };
```

Variadic templates

Variadic functions are functions with variable number of arguments.
C variadic function (runtime) : **printf("%s = %d\n", name, value);**
C variadic functions DO NOT WORK with C++ classes and references !

C++: variadic templates, ellipsis (**...**) is part of the C++ syntax!
The list of all arguments is called *variadic pack*.
Example: return the pack size (number of arguments):

```
template <typename... Args>
size_t packSize(const Args & ... pack){
        return sizeof...(pack);       // or
//      return sizeof...(Args);       // Same result
}
```

Arguments can be of *different types*.
For arguments of a single type, use **std::initializer_list** or **std::vector**.
Args : types of arguments. Ellipsis (**...**) is called *pack expansion*.

Example: print all arguments to cout

Non-variadic **print()** with 1 argument to break the recursion.

```
template <typename T>
void print(const T & t){
    cout << t << endl;
}
```

Variadic **print()** with recursion.

```
template <typename T, typename... Args>
void print(const T & t, const Args & ... rest){
    cout << t << endl;
    print(rest...);
}
```

Alternative to recursion: Initializer list:

vector<int> v{args...};

Works if all arguments are of the same type (or if you use **std::any**)

A simple C++ stream implementation of printf()

Use C-string **snprintf()** to write a fixed-size buffer.

```
template <typename... Params>
void printfCPP(std::ostream & os, const std::string & fmt, Params... p) {
    constexpr size_t SIZE = 1000;
    static char buffer[SIZE];  // Hidden global buffer = ugly
    std::snprintf(buffer, SIZE, fmt.c_str(), p...);
    os << buffer;
}
```

Usage example:

printfCPP(cout, "%s : %d * %d = %d\n", "Hello", 3, 7, 3*7);

Note : this function cannot print any classes or **std::string** !

Using global fixed-size buffer is BAD in real life!

Not thread-safe!

There are better versions, like **format()** in Boost.

Advanced template (meta)programming:

Compile-time factorial (from Wikipedia):

```
// Induction
template <int N>
struct Factorial {
  static const int value = N * Factorial<N - 1>::value;
};
// Base case via template specialization:
template <>
struct Factorial<0> {
  static const int value = 1;
};
```

Factorial is a class template with a single *static* field **value**.

This is a standard trick for *template metaprogramming*.

Usage:

```
cout << "Factorial<5>::value = " << Factorial<5>::value << endl;
```

Type traits: Various compile-time operations

is_pointer
remove_pointer
add_pointer
is_reference
remove_reference
add_rvalue_reference
is_class
is_integral
is_base_of
min
...

For example:

is_pointer<int*>::value	is **true** (of type **bool**)
is_pointer<int*>()	is an object of **true_type** (compile-time version of **bool**)
true_type::value	is **true** (of type **bool**)

is_pointer() example

Print either value or pointer : Naive approach (WRONG !) :

```
template<typename T>
void katana(const T & val){
        if (is_pointer<T>::value)
                cout << "Pointer : " << val << " , *val = "<< *val << endl;
        else
                cout << "Value : " << val << endl;
}
```

Problem: ***val** would not compile for a non-pointer !

We used here a regular (run-time) **if**, but we want instead some "compile-time **if**".

We cannot use the regular **if** for the template (compile-time) logic!

Solution: (C++ 17): **if constexpr** is the compile-time **if** (NOT preprocessor **#if** !):

```
if constexpr (is_pointer<T>::value)
        ...
```

is_pointer() example 2. Correct code for C++ < 17.

```
template<typename T>
void katanaImpl(const T & val, true_type){    // Pointer
    cout << "Pointer : " << val << " , *val = "<< *val << endl;
}

template<typename T>
void katanaImpl(const T & val, false_type){    // Not Pointer
    cout << "Value : " << val << endl;
}

template<typename T>                        // Print value or pointer
void katana(const T & val){
    katanaImpl(val, is_pointer<T>());   // true_type or false_type
}
```

is_pointer<T>() returns an object of either **true_type** or **true_type**.
One of the two overloaded templates **katanaImpl()** is selected.
This is called *tag dispatch* (second argument = tag).

Library of the day : TensorRT

How to infer pre-trained neural networks in C++?
There are many frameworks:

- frugally-deep : lightweight CPU-only, header-only
- Google: Tensorflow Lite (CPU/GPU)
- Facebook: libtorch (CPU/GPU)
- Microsoft: ONNX runtime (CPU/GPU)
- Nvidia: *TensorRT* (Nvidia GPU only !)

TensorRT Pros:
TensorRT is super-optimized for each particular GPU model, for inference only.
Supports float16, int8, tensor cores, Nvidia DLA.
Relatively lightweight (but requires CUDA). Also available for Jetson devices.
TensorRT Cons:
Incompatibilities between versions (5, 6, 7).
Not all common operations are supported (Image resize !).

How to port a PyTorch model to TensorRT ?

Let us create a trivial PyTorch model and export it to ONNX:

```
model = torch.nn.Linear(3, 2)                    # Linear layer (b, 3) -> (b, 2)
x = torch.randn(1, 3, requires_grad=True, device='cuda')  # sample input
torch.onnx.export(model,                         # model being run
                  x,                             # model sample input (correct size)
                  "model2.onnx",                 # where to save the model
                  verbose=True,                  # Verbose output
                  export_params=True,            # store the trained parameter weights
                  opset_version=11,              # ONNX version to export the model to
                  input_names=['input'],             # the model's input names
                  output_names=['output'],           # the model's output names
                  dynamic_axes={'input': {0: 'batch_size'},    # dynamic batch
                                'output': {0: 'batch_size'}})
```

Note : It is simpler to use a fixed batch size, but dynamic batch size is more interesting.

An ONNX model can be used in Microsoft ONNX runtime or converted to TensorRT.

See my complete TensorRT C++ examples at:

https://github.com/agrechnev/trt-cpp-min

Let us parse the ONNX model in C++ TensorRT

First, create the *builder*, which will later build an optimized TensorRT *engine*:

```
unique_ptr<IBuilder, Destroy<IBuilder>> builder{createInferBuilder(logger)};
```

Next, create a *network definition* (neural network not yet optimized):

```
unique_ptr<INetworkDefinition, Destroy<INetworkDefinition>> network{
    builder->createNetworkV2(1U << (unsigned)
    NetworkDefinitionCreationFlag::kEXPLICIT_BATCH)     };
```

Create a *parser* and parse the ONNX file into the *network definition*:

```
unique_ptr<nvonnxparser::IParser, Destroy<nvonnxparser::IParser>> parser{
        nvonnxparser::createParser(*network, logger)};
if (!parser->parseFromFile("model2.onnx",
        static_cast<int>(ILogger::Severity::kINFO)))
    throw runtime_error("ERROR: could not parse ONNX model !");
```

Create optimization profile, config, and build the engine (TensorRT >=6)

Create *optimization profile* and set the MIN/MAX/OPT values for the dynamic dimensions

```
IOptimizationProfile *profile = builder->createOptimizationProfile();
profile->setDimensions("input", OptProfileSelector::kMIN, Dims2{batchSize, 3});
profile->setDimensions("input", OptProfileSelector::kMAX, Dims2{batchSize, 3});
profile->setDimensions("input", OptProfileSelector::kOPT, Dims2{batchSize, 3});
```

Create the *config* and build the *engine* from *network def* (Optimization happens HERE):

```
unique_ptr<IBuilderConfig, Destroy<IBuilderConfig>>
        config(builder->createBuilderConfig());
config->addOptimizationProfile(profile);
unique_ptr<ICudaEngine, Destroy<ICudaEngine>> engine(
        builder->buildEngineWithConfig(*network, *config));
```

Create the *context* for the inference (can be multiple contexts/engine).
You must finally fix the dynamic dimensions here. Important!

```
unique_ptr<IExecutionContext, Destroy<IExecutionContext>>
        context(engine->createExecutionContext());
context->setBindingDimensions(0, Dims2(batchSize, 3));
```

TensorRT CPU memory management

TensorRT classes are *internal* (not part of the public API).

IBuilder etc. are *interfaces* : usable only via *polymorphism* (pointers or references).

For some reason, method **destroy()** is used instead of a virtual destructor.

We manage them with a **unique_ptr** :

```
unique_ptr<IBuilder, Destroy<IBuilder>> builder;
```

Where we use a *custom deleter* (a functor template)

```
template<typename T>
struct Destroy {
    void operator()(T *t) const {
        t->destroy();
    }
};
```

Classes without **destroy()** are managed by other objects (**ITensor**, **ILayer**, ...)

We do not use **unique_ptr** for them (would not compile if we tried) !

Running the inference

First, allocate GPU memory buffers for the input/output tensors

```
vector<float> inputTensor{3 * batchSize};
// Fill inputTensor with data
vector<float> outputTensor(2 * batchSize, -4.9);
void *bindings[2]{0};
cudaMalloc(&bindings[0], inputTensor.size() * sizeof(float));
cudaMalloc(&bindings[1], outputTensor.size() * sizeof(float));
```

Then, use the *context* to infer on a *cuda stream* :

```
cudaStream_t stream; cudaStreamCreate(&stream);
cudaMemcpyAsync(bindings[0], inputTensor.data(), inputTensor.size() *
        sizeof(float), cudaMemcpyHostToDevice, stream);
context->enqueueV2(bindings, stream, nullptr);
cudaMemcpyAsync(outputTensor.data(), bindings[1], outputTensor.size() *
        sizeof(float), cudaMemcpyDeviceToHost, stream);
cudaStreamSynchronize(stream);
// Don't forget to free the GPU memory afterwards !
cudaFree(bindings[0]); cudaFree(bindings[1]);
```

Saving and loading a TensorRT engine

Parsing ONNX and creating optimized engine can take a long time.

Solution: Save a TensorRT *engine* to a disk file.

```
unique_ptr<IHostMemory, Destroy<IHostMemory>>
        serializedEngine(engine->serialize());
ofstream out("example3.engine", ios::binary);
out.write((char *)serializedEngine->data(), serializedEngine->size());
```

Then, load the file and deserialize the engine.

Note: Engine only works for the current platform, GPU model and TensorRT version!

```
// Read the entire file to a buffer
vector<char> buffer;
ifstream in("example3.engine", ios::binary | ios::ate);
streamsize ss = in.tellg();
in.seekg(0, ios::beg);
buffer.resize(ss);
// Deserialize engine from the buffer
unique_ptr<IRuntime, Destroy<IRuntime>> runtime(createInferRuntime(logger));
unique_ptr<ICudaEngine, Destroy<ICudaEngine>>
    engine(runtime->deserializeCudaEngine(buffer.data(), buffer.size()));
```

Create TensorRT network definition by hand (no ONNX file parsing !)

Let us start with an empty *network definition*, add input (**ITensor**),
several layers (**ILayer** subclasses), and finally add the output (**ITensor**).
Each layer also can have inputs and outputs. Note : -1 is the dynamic dimension.
The operation is $\mathbf{y = x*W^T + b}$ (Linear layer).
Note : The ONNX parser uses similar operations under the hood.

```
ITensor *input = network->addInput("input", DataType::kFLOAT, Dims2(-1, 3));
IConstantLayer * const1 = network->addConstant(Dims2(2, 3), w);
IMatrixMultiplyLayer* mm = network->addMatrixMultiply(*input,
   MatrixOperation::kNONE, *const1->getOutput(0), MatrixOperation::kTRANSPOSE);
IConstantLayer * const2 = network->addConstant(Dims2(1, 2), b);
IElementWiseLayer * ew = network->addElementWise(*mm->getOutput(0),
   *const2->getOutput(0), ElementWiseOperation::kSUM);
ITensor *output = ew->getOutput(0);
output->setName("output");
network->markOutput(*output);
```

They really love computer vision in TensorRT!
They have all kind of convolution, max pool layers.
But no Linear layer! "Fully connected" works only for 4+ input dimensions !

Thank you for your attention !

title

text

C++ templates vs Java Generics (table)

C++	Java
Instantiated at compile time	Instantiated at run time
Functions, classes, methods	Classes, methods
Templates in .h files, not precompiled	Generics are precompiled
No class polymorphism	Based on a common parent (or Object)
Behavior can depend on type	Behavior is identical for all types
Can be used with primitive types	Only for class types
No simple way to limit type	Limit the type: extends, super, ?
Rich template metaprogramming	No metaprogramming
Turing complete language	No Turing complete language