Modern C++ for Computer Vision and Image Processing

Lecture 09: Templates

Ignacio Vizzo and Cyrill Stachniss

UNIVERSITÄT BONN

Generic programming

What is Programming?

- "The craft of writing useful, maintainable, and extensible source code which can be interpreted or compiled by a computing system to perform a meaningful task." —Wikibooks

What is Meta-Programming?

- "The writing of computer programs that manipulate other programs (or themselves) as if they were data." —Anders Hejlsberg

Meaning of template

Dictionary Definitions:

- Something that serves as a model for others to copy
- A preset format for a document or file
- Something that is used as a pattern for producing other similar things

Meaning of template

Static language (type check @ compile time) ಇಲ್ಲಿ Generic type ಇರುತ್ತೆ
(C++, Java, C#, . . .
Generic ಗೆ ಇನ್ನೊಂದು template ಅಂತಾರೆ.

C++ Definitions:

A template is a C++ entity that defines one of the following:

- A family of classes (class template), which may be nested classes.
- A family of functions (function template), which may be member functions.

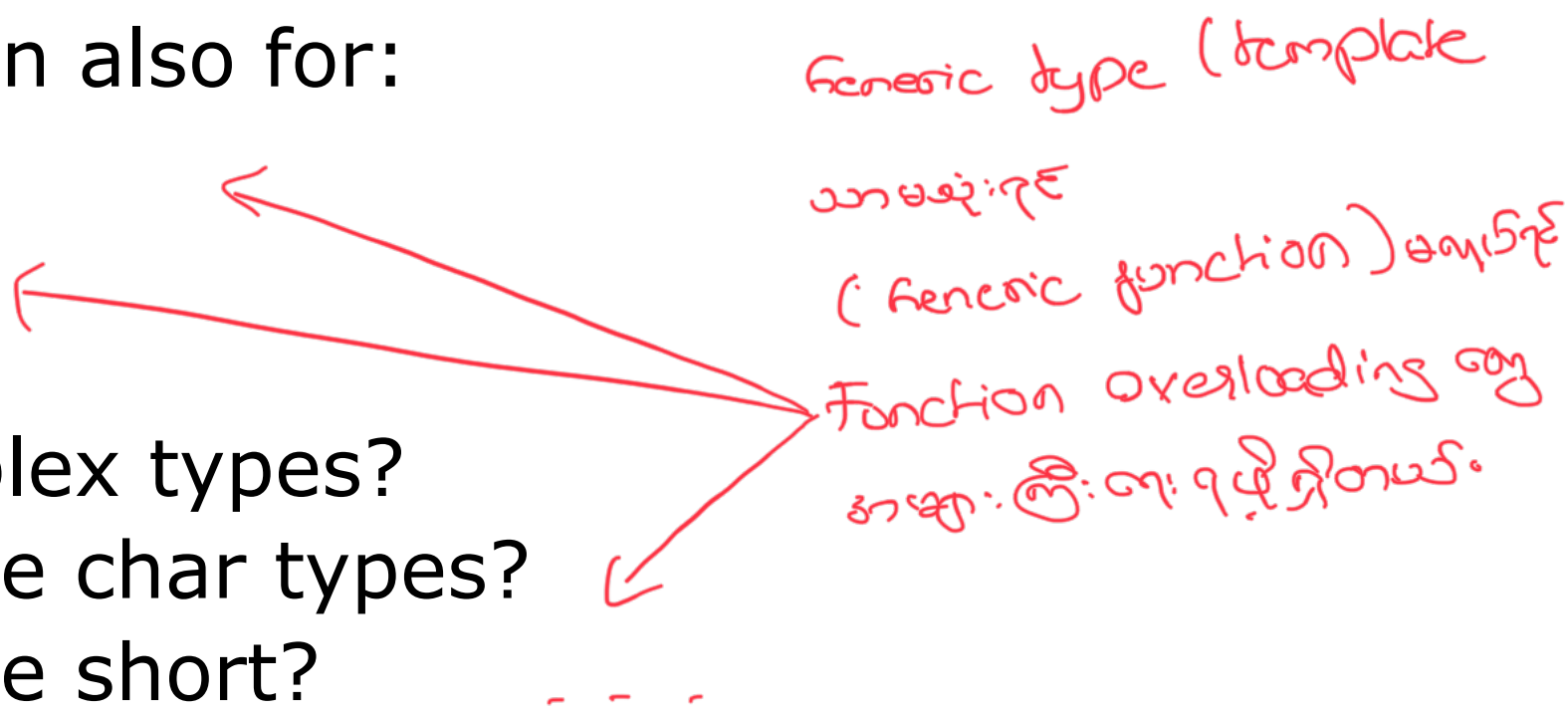
Motivation: Generic functions

abs():

```
1 double abs(double x) { return (x >= 0) ? x : -x; }
2    int abs(int x)    { return (x >= 0) ? x : -x; }
```

And then also for:

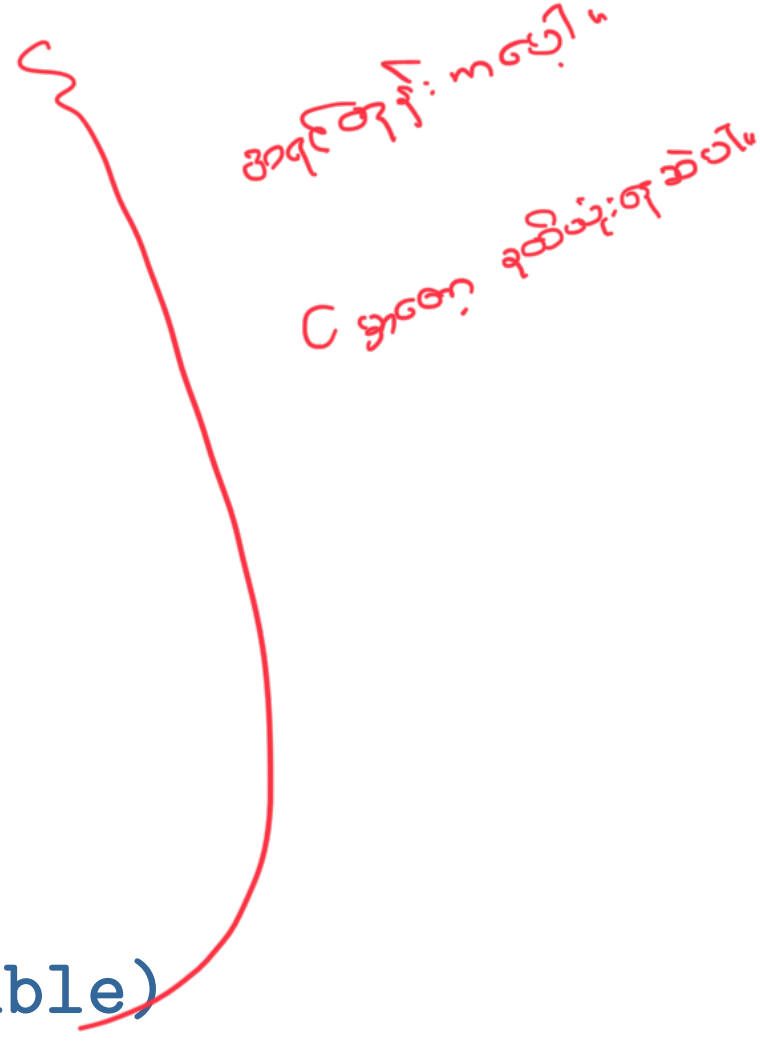
- long
- int
- float
- complex types?
- Maybe char types?
- Maybe short?
- Where does this end?



Motivation: Generic functions

C-style, C99 Standard:

- `abs (int)`
- `labs (long)`
- `llabs (long long)`
- `imaxabs (intmax_t)`
- `fabsf (float)`
- `fabs (double)`
- `fabsl (long double)`
- `cabsf (_Complex float)`
- `cabs (_Complex double)`
- `cabsl (_Complex long double)`



Function Templates

abs<T>():

```
1 template <typename T>
2 T abs(T x) {
3   return (x >= 0) ? x : -x;
4 }
```

- Function Template
  -
-
  - abs<int>

Template functions

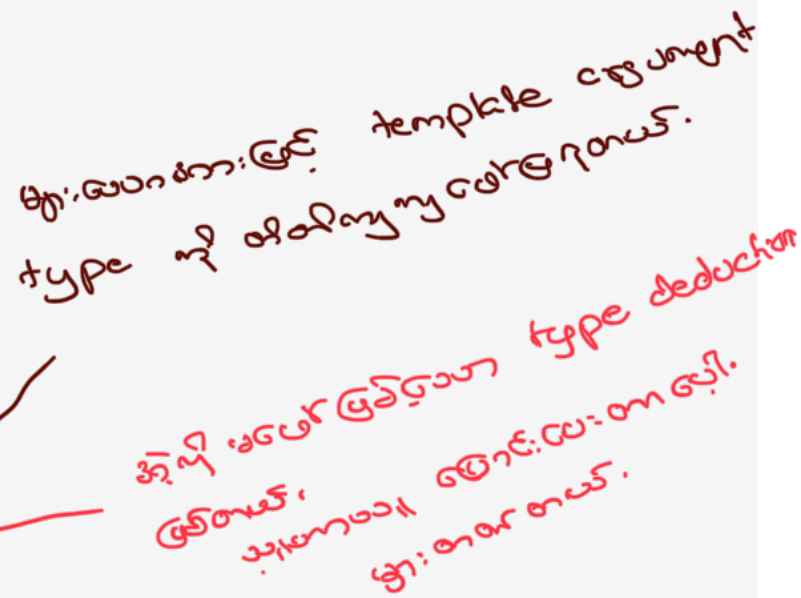
- Use keyword `template`

```
1 template <typename T, typename S>
2 T awesome_function(const T& var_t, const S& var_s) {
3   // some dummy implementation
4   T result = var_t;
5   return result;
6 }
```

- `T` and `S` can be any type.
- A **function template** defines a **family** of functions.

Using Function Templates

```
1  template <typename T>
2  T abs(T x) {
3    return (x >= 0) ? x : -x;
4  }
5
6  int main() {
7    const double x = 5.5;
8    const    int y = -5;
9
10   auto abs_x = abs<double>(x);
11   int  abs_y = abs<int>(y);
12
13   double abs_x_2 = abs(x);  // type-deduction
14   auto   abs_y_2 = abs(y);  // type-deduction
15 }
```



check overloading - function-template

Template classes

```
1 template <class T>
2 class MyClass {
3  public:
4   MyClass(T x) : x_(x) {}
5
6  private:
7   T x_;
8 };
```

- Classes templates are not classes.
  - **They are templates for making classes**
- Don’t pay for what you don’t use:
  - **If nobody calls MyClass<int>, it won’t be instantiated by the compiler at all.**

Template classes usage

typename vs class

template <template <--, --> class >

```
1  template <class T>
2  class MyClass {
3   public:
4    MyClass(T x) : x_(x) {}
5
6   private:
7    T x_;
8  };
9
10 int main() {
11   MyClass<int> my_float_object(10);
12   MyClass<double> my_double_object(10.0);
13   return 0;
14 }
```

Template Parameters

```
1 template <typename T, size_t N = 10>
2 T AccumulateVector(const T& val) {
3   std::vector<T> vec(val, N);
4   return std::accumulate(vec.begin(), vec.end(), 0);
5 }
```

- Every `template` is parameterized by one or more `template parameters`:
  `template < parameter-list > declaration`
- Think the `template parameters` the same way as any `function arguemnents`, but at **compile-time**.

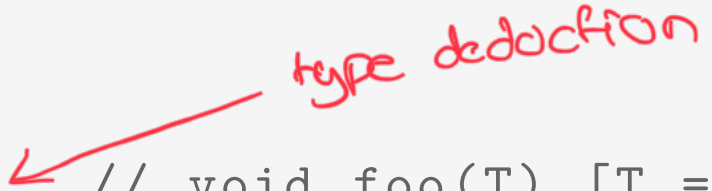
Template Parameters

```
1  template <typename T, size_t N = 10>
2  T AccumulateVector(const T& val) {
3    std::vector<T> vec(val, N);
4    return std::accumulate(vec.begin(), vec.end(), 0);
5  }
6
7  using namespace std;
8  int main() {
9    cout << AccumulateVector(1) << endl;
10   cout << AccumulateVector<float>(2) << endl;
11   cout << AccumulateVector<float, 5>(2.0) << endl;
12   return 0;
13 }
```

Type Deduction

Type deduction for function templates:

```
1  #include <cstdio>
2
3  template <typename T>
4  void foo(T x) {
5    puts(__PRETTY_FUNCTION__);
6  }
7
8  int main() {
9    foo(4);          // void foo(T) [T = int]
10   foo(4.2);        // void foo(T) [T = double]
11   foo("hello");    // void foo(T) [T = const char *]
12 }
```



Type Deduction Rules (short)

- Each function parameter may contribute (or not) to the deduction of each template parameter (or not).
- At the end of this process, the compiler checks to make sure that each template parameter has been deduced at least once (otherwise: `couldn't infer template argument T`) and that all deductions agree with each other (otherwise:`deduced conflicting types for parameter T`) .

Type Deduction

Type deduction for function templates:

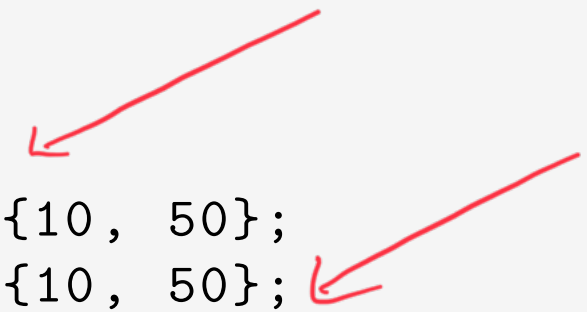
```
1  template <typename T, typename U>
2  void f(T x, U y) {
3    // ..
4  }
5  template <typename T>
6  void g(T x, T y)
7  // ..
8  }
9
10 int main() {
11   f(1, 2);  // void f(T, U) [T = int, U = int]
12   f(1, 2u); // void f(T, U) [T = int, U = unsigned int]
13   g(1, 2);  // void g(T, T) [T = int]
14   g(1, 2u); // error: no matching function for call
15             // to g(int, unsigned int)
16 }
```

05

Type Deduction

Type deduction for class templates:

```
1  template <typename T>
2  struct Foo {
3   public:
4    Foo(T x) : x_(x) {}
5    T x_;
6  };
7
8  int main() {
9    auto obj = Foo<int>(10).x_;
10   auto same_obj = Foo(10).x_;
11   auto vec = std::vector<int>{10, 50};
12   auto same_vec = std::vector{10, 50};
13 }
```



Note: New in C++17

Type Deduction Puzzle

```
1  template <typename T, typename U>
2  void foo(std::array<T, sizeof(U)> x,
3           std::array<U, sizeof(T)> y) {
4    puts(__PRETTY_FUNCTION__);
5  }
6
7  int main() {
8    foo(std::array<int, 8>{}, std::array<double, 4>{});
9    foo(std::array<int, 8>{}, std::array<double, 4>{});
10 }
```

Template Full Specialization

```
1  template <typename T>
2  bool is_void() {
3    return false;
4  }
5
6  template <>
7  bool is_void<void>() {
8    return true;
9  }
10
11 int main() {
12   std::cout << std::boolalpha
13             << is_void<int>() << std::endl
14             << is_void<void>() << std::endl;
15 }
```

Template Full Specialization

Template ಅಲ್ಲಿ Generic functions ಅನ್ನು
Generic class ಅನ್ನು ಸಹ
Data type ಅಷ್ಟೇ ಅಲ್ಲದೆ ನಿರ್ದಿಷ್ಟ ಬೇರೆ ಬೇರೆ
ಮಾಡಬಹುದು ಅಥವಾ data type ಅನುಸಾರ
behavior ಅನ್ನು ಬದಲಾಯಿಸಬಹುದು
definition ಅನ್ನು ಬದಲಾಯಿಸಬಹುದು
template param ಅನ್ನು ಬದಲಾಯಿಸಿದರೆ full specialization
ಮತ್ತು ಭಾಗಶಃ ಅನ್ನು ಬದಲಾಯಿಸಿದರೆ
partial template specialization ಆಗುತ್ತದೆ

Template Full Specialization

- Prefix the definition with `template<>`
- Then write the function **definition**.
- Usually means you don't need to write any more angle brackets at all.
- Unless `T` can't be deduced/**defaulted**:

```
1 template <typename T = void>
2 int my_sizeof() {
3   return sizeof(T);
4 }
5
6 template <>
7 int my_sizeof() {
8   return 1;
9 }
```

Template Partial Specialization

```
1  template <typename T>
2  constexpr bool is_array = false;
3
4  template <typename Tp>
5  constexpr bool is_array<Tp[]> = true;
6
7  int main() {
8    std::cout << std::boolalpha;
9    std::cout << is_array<int>   << std::endl   // false
10             << is_array<int[]> << std::endl;  // true
11 }
```

A partial specialization is any specialization that is, itself, a template. It still requires further "customization" by the user before it can be used.

Template headers/source

- Concrete templates are instantiated at compile time.
- Linker does not know about implementation
- There are three options for template classes:
  1. Declare and define in header files
  2. Declare in `NAME.hpp` file, implement in `NAME_impl.hpp` file, add `#include <NAME_impl.hpp>` in the end of `NAME.hpp`
  3. Declare in `*.hpp` file, implement in `*.cpp` file, in the end of the `*.cpp` add explicit instantiation for types you expect to use
- Read more about it:

http://www.drdobbs.com/moving-templates-out-of-header-files/184403420

Static code generatrion with constexpr

This specifier means rvalue or return rvalue is constant. (computed at compile time)

```
1  #include <iostream>
2  constexpr int factorial(int n) {
3    // Compute this at compile time
4    return n <= 1 ? 1 : (n * factorial(n - 1));
5  }
6
7  int main() {
8    // Guaranteed to be computed at compile time
9    return factorial(10);
10 }
```

- `constexpr` specifies that the value of a variable or function can appear in constant expressions

It only works if the variable of function can be defined at **compile-time**:

```
1  #include <array>
2  #include <vector>
3
4  int main() {
5    std::vector<int> vec;
6    constexpr size_t size = vec.size();  // error
7
8    std::array<int, 10> arr;
9    constexpr size_t size = arr.size();  // works!
10 }
```

error: constexpr variable 'size' must be initialized by a constant expression

C++11

లాంబ్డా: reuse చేయడానికి (function name లేకుండా) ఉన్న inline function (code చిన్నగా రాయడానికి) పేరే lambda ని లాంబ్డా ఫంక్షన్లు అంటారు.

[capture clause] (params) -> return_type { definition }

```
[] () {};
```

capture list — function arguments — function body

How to name lambda function?

```
auto my_func = [] () { cout << "hello"; };

my_func();
```