HIGH-LEVEL PROGRAMMING 2

Function Templates

by Prasanna Ghali

Function Templates: Motivation (1/9)

- *Function* is our fundamental programming unit
- Function such as `sqrt` allows us to design and implement an algorithm once and reuse it

$$x \longrightarrow \boxed{\texttt{sqrt}} \longrightarrow \sqrt{x}$$

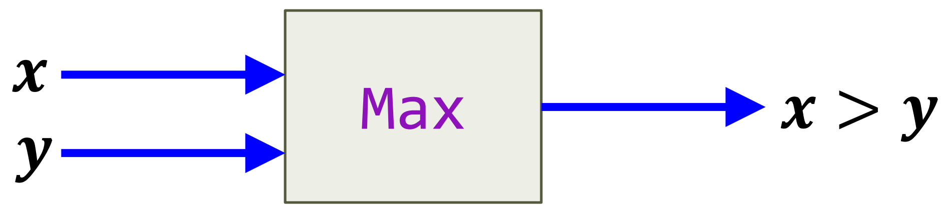
- To reuse function, we *parameterize* the function to take values of specific type:

$$x\ (\texttt{double}) \longrightarrow \boxed{\texttt{sqrt}} \longrightarrow \sqrt{x}\ (\texttt{double})$$

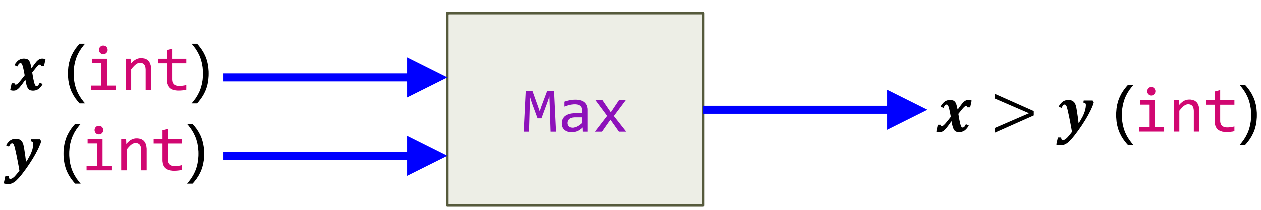
```
double sqrt(double x) {
  // implementation algorithm to return square root of x
}
```

Function Templates: Motivation (2/9)

- Suppose we want to write function `Max(x, y)` where `x` and `y` are expressions of same type



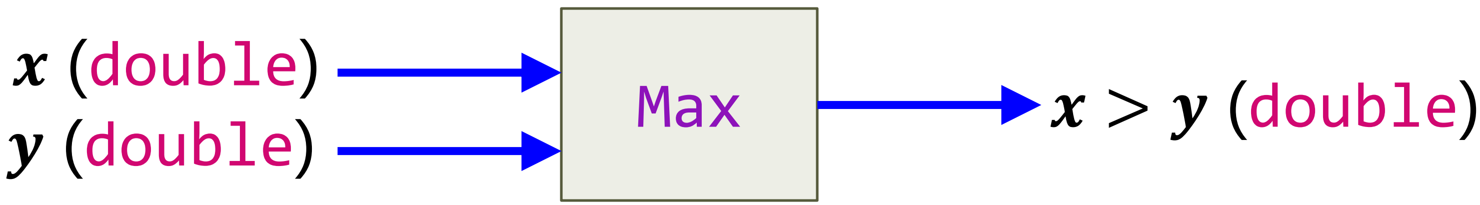
- To reuse function `Max` for `int` values, we parameterize the function:




```
int Max(int lhs, int rhs) {
  return lhs > rhs ? lhs : rhs;
}
```

Function Templates: Motivation (3/9)

- To reuse function `Max` for `double` values, we parameterize the function like this:




```
double Max(double lhs, double rhs) {
  return lhs > rhs ? lhs : rhs;
}
```

Function Templates: Motivation (4/9)

- Using function overloading, we can do this:

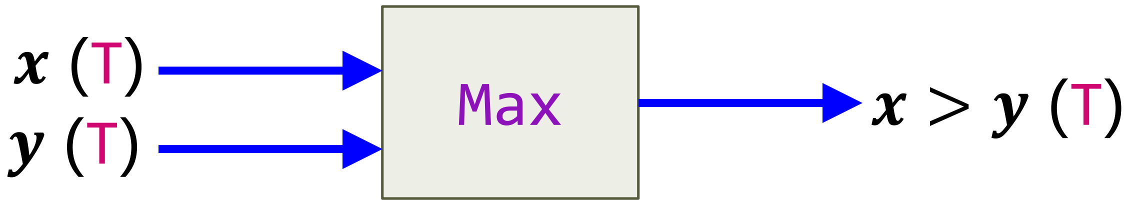
```
int Max(int lhs, int rhs) {
  return lhs > rhs ? lhs : rhs;
}

double Max(double lhs, double rhs) {
  return lhs > rhs ? lhs : rhs;
}
```

Function Templates: Motivation (5/9)

- There're lots of other built-in numeric types for which function `Max` must be overloaded:
  - `char`, `short`, `long`
  - `unsigned` counterparts of above types
  - Other floating-point types such as `float`, `long double`
- There are more types for which `Max` must be overloaded:
  - User-defined types as `Point`, `Vector`, `Matrix`, …
  - Containers such as `std::string`, `std::vector<T>`, `std::forward_list<T>`, `std::list<T>`, …

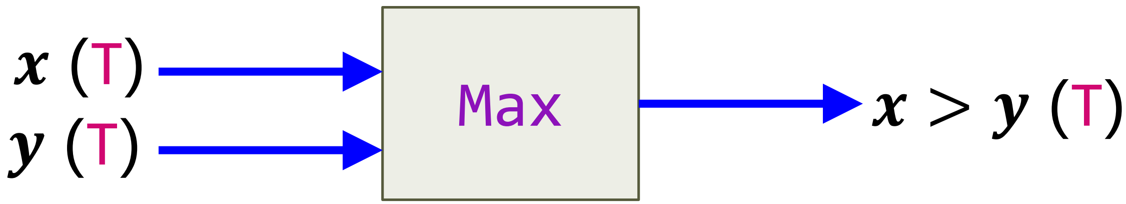
Function Templates: Motivation (6/9)

- Since function `Max(x, y)` [where `x` and `y` are expressions of same type] is similar for all types, why can't compiler generate function overloads for necessary types?
- That is, why not take concept of parameterization a step further to allow `Max(x, y)` to be *parameterized on type* of `x` and `y`?

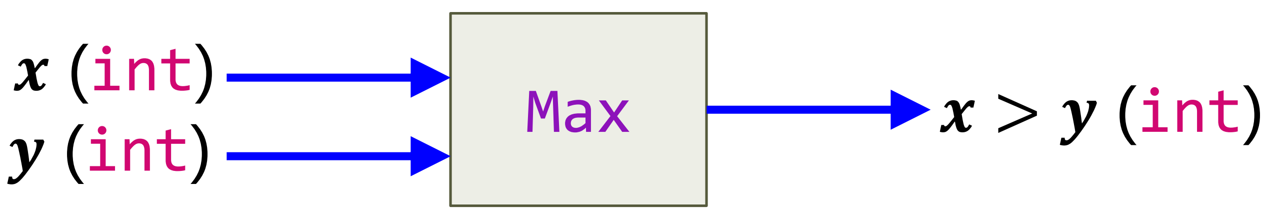


Function Templates: Motivation (7/9)

- Given:

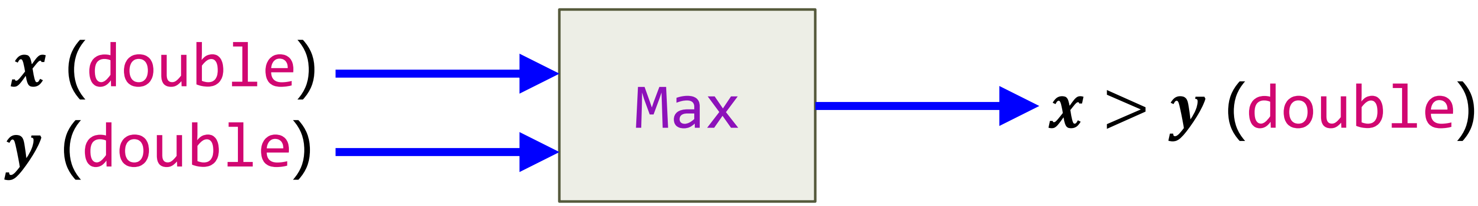


- If we say `Max(10, 11)` compiler will automatically instantiate:

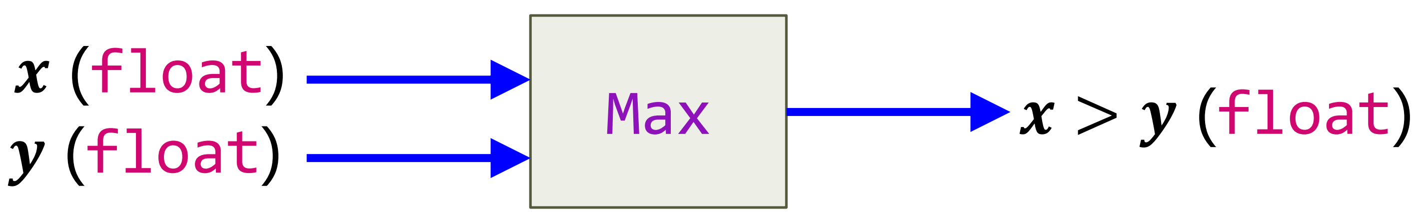


Function Templates: Motivation (8/9)

- If we say `Max(10.1, 11.1)` compiler will automatically instantiate:



- If we say `Max(10.1f, 11.1f)` compiler will automatically instantiate:



Function Templates: Motivation (9/9)

- If we say

```
std::string sin{"Singapore"}, sea{"Seattle"};
Max(sin, sea);
```

compiler will automatically instantiate

x (std::string) → Max → $x > y$ (std::string)
y (std::string) →

Function Templates: Definition

- Function template provides blueprint for compiler to generate potentially infinite number of function overloads

2 Keyword `template` identifies this code as a template

3 Keyword `typename` or `class` identifies T as type

4 T is called *template type parameter* signifying that type of function parameters `lhs` and `rhs` is left open

5 Every occurrence of T is replaced by an actual type when an instance of template is created

```
template <typename T> T Max(T lhs, T rhs)
{
  return lhs > rhs ? lhs : rhs;
}
```

1 This function template specifies *family* of functions that return maximum of two values of *same type*, which are passed as function parameters `lhs` and `rhs`

Using Function Template


```
template <typename T>
T Max(T lhs, T rhs) {
  return lhs > rhs ? lhs : rhs;
}

int main() {
  int i1{21}, i2{42};
  std::cout << Max(i1, i2) << '\n';

  double d1{3.14}, d2{2.13};
  std::cout << Max(d1, d2) << '\n';

  std::string s1{"enjoy"}, s2{"enjoyment"};
  std::cout << Max(s1, s2) << '\n';
}
```

Step 1: Template Argument Deduction

- Consider function template and call to that function template:

```
// function template declaration
template <typename T>
T Max(T lhs, T rhs);

// Max is called with int arguments
Max(10, 12);
```

- *Template argument deduction* is process during compilation when compilers use function call arguments to deduce template argument type
  - Here, template argument type is deduced as `int`

Step 2: Template Instantiation

- Using TAD, `int` is deduced as template argument type:

```
// function template declaration
template <typename T>
T Max(T lhs, T rhs);

// int is template argument type
Max(10, 12);
```

- Compiler will replace template parameter `T` with concrete type `int` to create this template instance:

```
// function template instance with T = int
int Max<int>(int lhs, int rhs) {
  return lhs > rhs ? lhs : rhs;
}
```

Step 3: Call to Template Instantce

- Consider function template and call to that function template:

```
// function template declaration
template <typename T>
T Max(T lhs, T rhs);

// int is template argument type
Max(10, 12);
```

- Compiler will replace call `Max(10, 12)` with `Max<int>(10, 12)`:

```
// function template instance with T = int
int Max<int>(int lhs, int rhs) {
  return lhs > rhs ? lhs : rhs;
}
// call function template instance with T = int
Max<int>(10, 12);
```

Function Templates: Terminology

T is called *template type parameter* signifying that type of function parameters lhs and rhs is left open

1

lhs and rhs are function *call parameters*

2

```
template <typename T> T Max(T lhs, T rhs) {
  return lhs > rhs ? lhs : rhs;
}

Max<int>(10, 12);
```

4

3

lhs and rhs are function *call arguments*

This is called *template type argument*. *Template argument deduction* is automatic process during compilation where call arguments are compared with call parameters to deduce template type argument.

Two-Phase Translation (1/2)

- Following code will not compile!!!

```cpp
#include <iostream>
#include <complex>

template <typename T>
T Max(T lhs, T rhs) {
  return lhs > rhs ? lhs : rhs;
}

int main() {
  // ok: operator > defined for int
  std::cout << Max(10, 11) << '\n';

  std::complex<double> c1{1.0, 2.0}, c2{2.0, 3.0};
  // error!!! operator > is not defined for class complex<double>
  std::complex<double> c3 = Max(c1, c2);
}
```

Two-Phase Translation (2/2)

- Thus, templates are compiled in two phases:
- First, without instantiation at definition time, template code itself is checked for correctness ignoring template type parameters
  - Syntax errors [such as missing semicolons] are discovered
  - Discovering use of unknown names [type names, function names, …] that don't depend on template type parameters
- At instantiation time, instantiated code is checked to ensure all code is valid [that is, all parts that depend on template type parameters are double-checked]

Compiling and Linking (1/2)

- Two-phase translation leads to important problem in handling of templates in practice:
  - When function template is used in way that triggers its instantiation, compiler will need to see that template's definition [not just its declaration]
- This *breaks* usual compile and link distinction for ordinary functions, when function declaration is sufficient to compile its use

Compiling and Linking (2/2)

- Simplest and most common approach: *Define function templates in header files that provide include guards*

```cpp
// some source file ...
#include <iostream>
#include "max.hpp"

int main() {
  std::cout << Max(10, 11) << '\n';
  std::cout << Max(10.1, 11.2) << '\n';
}
```

```cpp
// in max.hpp ...
#ifndef MAX_HPP_
#define MAX_HPP_

template <typename T>
T Max(T lhs, T rhs) {
  return lhs > rhs ? lhs : rhs;
}

#endif
```

Will Following Code Compile? (1/4)

- Given function template

```
template <typename T>
T Max(T lhs, T rhs);
```

- Will following calls to `Max` compile?

```
int i{10}, &ri{i}, *pi{&i}, ai[4]{1,2,3,4};
int const ci{42};

Max(i, ci);  // 1
Max(ci, ci); // 2
Max(i, ri);  // 3
Max(i, ci);  // 4
Max(pi, ai); // 5 [not recommended!!!]
```

Will Following Code Compile? (2/4)

- Given function template

```
template <typename T>
T Max(T lhs, T rhs);
```

- Will following calls to `Max` compile?

```
int i{10};
double d{11.2};
Max(i, d); // 1
```

```
std::string s{"enjoyment"};
Max("enjoy", s); // 2
```

Will Following Code Compile? (3/4)

- Will following calls to Max compile?

```
template <typename T>
T Max(T lhs, T rhs);

int i{10}, &ri{i}, *pi{&i}, ai[4]{1,2,3,4};
int const ci{42};

Max(i, ci);  // Yes: T = int
Max(ci, ci); // Yes: T = int
Max(i, ri);  // Yes: T = int
Max(ri, ci); // Yes: T = int
Max(pi, ai); // Yes: T = int*
```

Will Following Code Compile? (4/4)

- Will following calls to Max compile?

```
template <typename T>
T Max(T lhs, T rhs);

int i{10};
double d{11.2};
Max(i, d); // NO: T can be deduced
           // as int or double

std::string s{"enjoyment"};
Max("enjoy", s); // NO: T can be deduced as
                 // char const[6] or std::string
```

Type Conversions During TAD (1/4)

- Type conversions are limited during TAD!!!
- When declaring call parameters by value, only trivial conversions that *decay* are supported for call arguments:
  - Top level `const` and `volatile` specifiers are ignored
  - References convert to referenced type
  - Raw arrays or functions convert to corresponding pointer type
  - *Decayed* types of 2 arguments must match exactly in call to function with 2 parameters declared with same template type parameter `T`

Type Conversions During TAD (2/4)


```
template <typename T>
T Max(T lhs, T rhs);

int i{10}, &ri{i}, *pi{&i}, ai[4]{1,2,3,4};
int const ci{42};

Max(i, ci);  // Yes: T = int
Max(ci, ci); // Yes: T = int
Max(i, ri);  // Yes: T = int
Max(ri, ci); // Yes: T = int
Max(pi, ai); // Yes: T = int*
```

Type Conversions During TAD (3/4)


```
template <typename T>
T Max(T lhs, T rhs);

int i{10};
double d{11.2};
Max(i, d); // NO: T can be deduced as int or double
           // with neither argument converted to other

std::string s{"enjoyment"};
Max("enjoy", s); // NO: T can be deduced as
                 // char const* or std::string!!!
                 // with neither argument converted to other
```

Type Conversions During TAD (4/4)

- Type conversions are limited during TAD!!!
- When declaring call parameters by value, only trivial conversions that *decay* are supported for call arguments
- When declaring call parameters by reference, *even trivial conversions don't apply*
  - Two arguments must match exactly in call to function with two parameters declared with same template type parameter T

Implicit Template Instantiation

- Earlier discussion showed that TAD allows us to call function templates with syntax similar to that of calling ordinary function
- Since TAD is automatically done by compiler, this is also known as *implicit instantiation*

```
int i{10}, &ri{i}, *pi{&i}, ai[4]{1,2,3,4};
int const ci{42};

Max(i, ci);  // Yes: T = int
Max(ci, ci); // Yes: T = int
Max(i, ri);  // Yes: T = int
Max(ri, ci); // Yes: T = int
Max(pi, ai); // Yes: T = int*
```

```
template <typename T>
T Max(T lhs, T rhs);
```

Handling Template Instantiation Error

- However, compiler will determine the following as error

```
template <typename T> T Max(T lhs, T rhs);
Max(10, 11.2);
```

- Three ways to handle such errors:
  - Cast argument(s) so that both match

    ```
    Max(static_cast<double>(10), 11.2);
    ```

  - Specify *explicitly* the type of T to prevent compiler from attempting TAD!!!
  - Specify that call parameters will have different types [more on this later]

Explicit Template Instantiation

- For *explicit template instantiation*, put explicit type argument for function template between angle brackets after function name

```
// generate instance with T as type double followed by
// implicit conversion of 1st argument 10 to 10.0
Max<double>(10, 11.2);
```

- Compiler has complete faith that you know what you're doing!!!

```
// generate instance with T as type int with implicit
// conversion of 1st argument from long double
// to int and 2nd argument from double to int
Max<int>(10.1L, 11.2);
```

Multiple Template Parameters (1/5)

- You could define `Max` template for call parameters of two potentially different types

```
template <typename T1, typename T2>
T1 Max(T1 lhs, T2 rhs) {
  return lhs > rhs ? lhs : rhs;
}

// OK but type of 1st argument defines return type
int i = Max(4, 7.2);
```

Multiple Template Parameters (2/5)

- What happens with following calls?
- Major problem is that return type depends on call argument order: maximum of `66.66` and `42` will be `double` `66.66` while maximum of `42` and `66.66` will be `int` `66`

```
template <typename T1, typename T2>
T1 Max(T1 lhs, T2 rhs) {
  return lhs > rhs ? lhs : rhs;
}

std::cout << Max(66.66, 42) << '\n'; // 1
std::cout << Max(42, 66.66) << '\n'; // 2
```

Multiple Template Parameters (3/5)

- C++ provides different ways to deal with the problem:
  - Introduce 3rd template parameter for return type
  - Let compiler find out return type
  - Declare return type to be "common type" of two parameter types

```
template <typename T1, typename T2>
T1 Max(T1 lhs, T2 rhs) {
  return lhs > rhs ? lhs : rhs;
}

std::cout << Max(66.66, 42) << '\n'; // 1
std::cout << Max(42, 66.66) << '\n'; // 2
```

Multiple Template Parameters (4/5)

- You can introduce 3rd template type parameter to define return type
  - TAD cannot deduce return type since it doesn't appear in call arguments: `Max(42, 66.66); // RT = ???`
  - As consequence, you've to specify template argument list explicitly

```
template <typename T1, typename T2, typename RT>
RT Max(T1 lhs, T2 rhs) {
  return lhs > rhs ? lhs : rhs;
}
```

```
Max<int, double, double>(42, 66.66); // OK but tedious
Max<double, int, double>(66.66, 42); // OK but tedious
```

Multiple Template Parameters (5/5)

- Better approach is to specify only 1st argument explicitly and to allow TAD to derive the rest

```
template <typename RT, typename T1, typename T2>
RT Max(T1 lhs, T2 rhs) {
  return lhs > rhs ? lhs : rhs;
}

Max<double>(42, 66.66); // OK: return type is double,
                        // T1 and T2 are deduced
Max<double>(66.66, 42); // OK: return type is double,
                        // T1 and T2 are deduced
```

Deducing Return Type (1/2)

- C++ provides different ways to deal with the problem [shown in code fragment]:
  - Introduce 3rd template parameter for return type
  - Let compiler find out return type
  - Declare return type to be "common type" of two parameter types

```
template <typename T1, typename T2>
T1 Max(T1 lhs, T2 rhs) {
  return lhs > rhs ? lhs : rhs;
}

// OK but type of 1st argument defines return type
int i = Max(4, 7.2);
```

Deducing Return Type (2/2)

- Since C++14, simplest and best approach to deduce return type is to let compiler find out
- How this magic works will be explained in HLP3

```
template <typename T1, typename T2>
auto Max(T1 lhs, T2 rhs) {
  return lhs > rhs ? lhs : rhs;
}

Max(42, 66.66); // OK: return type is double and is
                // deduced from return statement
Max(66.66, 42); // OK: return type is double and is
                // deduced from return statement
```

Return Type as Common Type (1/2)

- C++ provides different ways to deal with the problem [shown in code fragment]:
  - Introduce 3rd template parameter for return type
  - Let compiler find out return type
  - Declare return type to be "common type" of two parameter types

```
template <typename T1, typename T2>
T1 Max(T1 lhs, T2 rhs) {
  return lhs > rhs ? lhs : rhs;
}

// OK but type of 1st argument defines return type
int i = Max(4, 7.2);
```

Return Type as Common Type (2/2)

- Since C++11, standard library provides means to specify choosing "most general type"
- `std::common_type_t<>` yield "common type" of two [or more] different types passed as template parameters [type traits will be explained in HLP3]

```
template <typename T1, typename T2>
std::common_type_t<T1,T2> Max(T1 lhs, T2 rhs) {
  return lhs > rhs ? lhs : rhs;
}

Max(42, 66.66); // OK: return type is double and is
                // most common type of int and double
Max(66.66, 42); // OK: return type is double and is
                // most common type of int and double
```

Default Values for Template Parameters (1/3)

- You can specify `double` as default return type in previous example

```
template <typename RT=double, typename T1, typename T2>
RT Max(T1 lhs, T2 rhs) {
  return lhs > rhs ? lhs : rhs;
}

Max(42, 66.66); // OK: return type is double,
                // T1 and T2 are deduced
Max(66.66, 42); // OK: return type is double,
                // T1 and T2 are deduced
```

Default Values for Template Parameters (2/3)

- You've some options – here, we've function template with default value referring to earlier parameter

```
// example is not very practical!!!
template <typename T1, typename T2, typename RT=T2>
RT Max(T1 lhs, T2 rhs) {
  return lhs > rhs ? lhs : rhs;
}

Max(42, 66.66); // OK: T1 and T2 are deduced
                // RT is double
Max(66.66, 42); // OK: T1 and T2 are deduced
                // RT is int
```

Default Values for Template Parameters (3/3)

- You can also use `std::common_type_t<>` to specify default value for return type

```
template <typename T1, typename T2,
          typename RT=std::common_type_t<T1,T2>>
RT Max(T1 lhs, T2 rhs) {
  return lhs > rhs ? lhs : rhs;
}

Max(42, 66.66); // OK: return type is double and is
                // most common type of int and double
Max(66.66, 42); // OK: return type is double and is
                // most common type of int and double
```

Problem With Current Version of Max

- Is there a better way to define function template Max?

```
// in max.hpp ...
#ifndef MAX_HPP_
#define MAX_HPP_

template <typename T>
T Max(T lhs, T rhs) {
  return lhs > rhs ? lhs : rhs;
}

#endif
```

- Hint: *Remember we're parameterizing a function* *Max(x, y) to be parameterized on potentially infinite types of x and y*

Better Version of Max

- Max can be called on types defined in program and types yet to be invented!!!
- From efficiency perspective, Max must be defined to take const references!!!

```
// in max.hpp ...
#ifndef MAX_HPP_
#define MAX_HPP_

template <typename T>
T Max(T const &lhs, T  const &rhs) {
  return lhs > rhs ? lhs : rhs;
}

#endif
```

Template Nontype Parameters (1/5)

- Templates can also have nontype parameters that require nontype arguments
- Suppose you need function to perform range checking on value, where range limits are fixed

```
// nontype parameters limited to integral types:
// integral numbers, bool, and pointers
template <typename T, int LOW, int HIGH>
bool is_in_range(T const& val) {
  return LOW <= val && val <= HIGH;
}

is_in_range(100); // ERROR: LOW and HIGH are unspecified
is_in_range<int, -50, 10>(42);      // OK: 0
is_in_range<double, 0, 500>(66.66); // OK: 1
```

Template Nontype Parameters (2/5)

- Better to put template nontype parameters to left of template type parameter
  - Allows compiler to deduce type argument

```
template <int LOW, int HIGH, typename T>
bool is_in_range(T const& val) {
  return LOW <= val && val <= HIGH;
}

is_in_range<-50, 10>(42);    // OK: 0 with T=int
is_in_range<0, 500>(66.66); // OK: 1 with T=double
```

Template Nontype Parameters (3/5)

- Following function template defines group of functions for which a certain value can be added:

```
template <int VAL, typename T>
T add_value(T x) {
  return x + VAL;
}
```

Template Nontype Parameters (4/5)

- You can use function template `add_value<>` to add any `int` value to every element of any array:

```
template <typename T, typename Func>
T* xform(T *first, T *last, Func fptr) {
  while (first != last) {
    *first = fptr(*first);
    ++first;
  }
  return first;
}

int ai[5] {1, 2, 3, 4, 5};
xform(ai, ai+5, add_value<3, int>);

double ad[4] {1.1, 2.2, 3.3, 4.4};
xform(ad, ad+4, add_value<5, double>);
```

Template Nontype Parameters (5/5)

- In C++17, nontype parameter may be integral type or pointer [reference] to an object or to function type
- In all cases, nontype parameter must be constant expression [known at compile-time]

Function Templates with Raw Array Arguments (1/8)

- What happens in following code fragment?

```
template <T>
void f(T param);

int ai[4] {1,2,3,4};
int const cai[4] {1,2,3,4};

f(ai);  // T = ??? and param = ???
f(cai); // T = ??? and param = ???
```

Function Templates with Raw Array Arguments (2/8)

- We know that functions can't declare parameters that are *truly* arrays
  - Array name *decays* to pointer to first element of array

```
template <T>
void f(T param);

int ai[4] {1,2,3,4};
int const cai[4] {1,2,3,4};

f(ai);  // T = int* and param = int*
f(cai); // T = int const* and param = int const*
```

Function Templates with Raw Array Arguments (3/8)

- When passing raw array by value, the type decays to pointer

```
template <typename T>
T sum(T const *first, T const *last) {
  T total{*first++};
  while (first != last) {
    total += *first++;
  }
  return total;
}

int ai[5] {1, 2, 3, 4, 5};
sum(ai, ai+5);
```

Function Templates with Raw Array Arguments (4/8)

- Although functions can't declare parameters that are truly arrays, they can declare parameters that are *references* to arrays

```
template <T>
void g(T& param);

int ai[4] {1,2,3,4};
int const cai[4] {1,2,3,4};

g(ai);  // T = ??? and param = ???
g(cai); // T = ??? and param = ???
```

Function Templates with Raw Array Arguments (5/8)

- Although functions can't declare parameters that are truly arrays, they can declare parameters that are *references* to arrays

```
template <T>
void g(T& param);

int ai[4] {1,2,3,4};
int const cai[4] {1,2,3,4};

g(ai);  // T = int[4] and param = int[4]
g(cai); // T = const int[4] and param = const int[4]
```

Function Templates with Raw Array Arguments (6/8)

- When passing raw array by reference, the type *does not decay* to pointer

```
double average5(double const (&rad)[5]) {
  double total{};
  for (double x : rad) {
    total += x;
  }
  return total/static_cast<double>(5.0);
}

double ad[5] {1.1, 2.2, 3.3, 4.4, 5.5};
average5(ad);
```

Function Templates with Raw Array Arguments (7/8)

- However, how to write function template to compute sum of raw array elements?

```
template <typename T>
??? sum(T &param) {
  ??? total{};
  for (??? const& x : param) {
    total += x;
  }
  return total;
}

double ad[5] {1.1, 2.2, 3.3, 4.4, 5.5};
sum(ad);
```

Function Templates with Raw Array Arguments (8/8)

- Function templates with nontype parameters can pass raw arrays of both varying sizes and varying element types

```
template <typename T, size_t N>
T average(T const (&rad)[N]) {
  T sum{};
  for (T const& x : rad) {
    sum += x;
  }
  return sum/N;
}

double ad[5] {1.1, 2.2, 3.3, 4.4, 5.5};
average(ad);
```

Review

- What is a function template?
- What are template type and nontype parameters?
- What are template arguments?
- What is template argument deduction?
- What is implicit and explicit template instantiation?
- Multiple template parameters
- Default template parameters