Slide 1

Notation of Parameters

Notation of the types of the parameters to a function can be written in several ways:

```
void foo(const int &x);
void foo(const int& x);
void foo(int const &x);
void foo(int const& x);
```

YOU have chosen the last shown notation, because it is easier to read.

Slide 2

Notation of Parameters . . . *(continued)*

The notation

```
void foo(int const& x);
```

makes it easy to see that the parameter is of type `int`, and that that is a `const` reference parameter, and finally that its name is `x`.

So, this is the way parameters are specified in this course!

Slide 3

Function Templates

Let the function `max` be defined as follows

```
int const& max(int const& a, int const& b)
{
    return a < b ? b : a;
}
```

What if you want a function `max` that takes parameters of type `float`?

Then you have to write a new identical function, just with parameters of type `float`.

Any solution? Yes, Function Templates.

Slide 4

Function Templates . . . *(continued)*

A *Function Template* is defined as follows

```
template <typename T>
T const& max(T const& a, T const& b)
{
    return a < b ? b : a;
}
```

where the symbol `T` can be any type you like, i.e. `int`, `float`, `double`, or `your_own_type` — (defined by the user).

Slide 5

Function Templates ... *(continued)*

The definition

```
template <typename T>
T const& max(T const& a, T const& b)
{
    return a < b ? b : a;
}
```

means that the function `max` can only be used for *one* arbitrary type `T`, i.e. the types of the parameters and the return type must be the same.

Slide 6

Using the Template

```
#include <iostream>
#include <string>
#include "max.h"

int main()
{
    int i = 42;
    std::cout << "max(7,i):    "
              << ::max(7,i) << std::endl;
```

Slide 7

```
    double f1 =  3.4;
    double f2 = -6.7;
    std::cout << "max(f1,f2):  "
              << ::max(f1,f2) << std::endl;

    std::string s1 = "mathematics";
    std::string s2 = "math";
    std::cout << "max(s1,s2):  "
              << ::max(s1,s2) << std::endl;

    return 0;
}
```

Slide 8

Output from the Program

The function `max` is called three times: once for two `int`s, once for two `double`s, and once for two `std::string`s.

The output from the program is

```
max(7,i):   42
max(f1,f2): 3.4
max(s1,s2): mathematics
```

Slide 9

Argument Deduction

```
template <typename T>
T const& max(T const& a, T const& b)
{
    return a < b ? b : a;
}
...
::max(4,7);     // OK: T is int
                // for both arguments
::max(4,4.2);   // ERROR: first T is int,
                // second T is double
```

Slide 10

Argument Deduction . . . *(continued)*

There are three ways to handle such an error:

1. Cast the arguments:

   ```
   ::max(static_cast<double>(4),4.2);
   ```

2. Specify explicitly the type `T`

   ```
   ::max<double>(4,4.2);
   ```

3. Specify that the parameters may have different types.

Slide 11

Template Parameters

Function templates have two kind of parameters:

1. *Template parameters*

   ```
   template <typename T>
   ```

   `T` is a template parameter.

2. *Call parameters*

   ```
   ... max(T const& a, T const& b)
   ```

   `a` and `b` are call parameters.

Slide 12

Template Parameters . . . *(continued)*

```
template <typename RT, typename T1, typename T2>
RT max(T1 const& a, T2 const& b)
{
    return a < b ? static_cast<RT>(b)
                 : static_cast<RT>(a);
}
```

Because of the type conversion a *value* is returned – not a *const reference.*

```
::max<double>(4,4.2); // OK: return type double
```

Slide 13

Overloading Function Templates

```
int const& max(int const& a, int const& b)
{
    return a < b ? b : a;
}

template <typename T>
T const& max(T const& a, T const& b)
{
    return a < b ? b : a;
}
```

Slide 14

```
template <typename T>
T const& max(T const& a, T const& b, T const& c)
{
    return ::max(::max(a, b), c);
}
```

Slide 15

Overloading Function Templates ...

```
int main()
{
    ::max(7, 42, 68);
    ::max(7.0, 42.0);
    ::max('a', 'b');
    ::max(7, 42);
    ::max<>(7, 42);
    ::max<double>(7, 42);
    ::max('a', 42.7);
    return 0;
}
```

Slide 16

Overloading Function Templates ...

`::max(7, 42, 68);` calls the template for three arguments

`::max(7.0, 42.0);` calls `max<double>`
— (by argument deduction)

`::max('a', 'b');` calls `max<char>`
— (by argument deduction)

`::max(7, 42);` calls *nontemplate* for two `ints`

`::max<>(7, 42);` calls `max<int>` (by argument deduction)

`::max<double>(7, 42);` calls `max<double>`
— (no argument deduction)

`::max('a', 42.7);` calls the *nontemplate* for two `ints`

Overloading Function Templates ...

Slide 17

```
#include <iostream>
#include <cstring>
#include <string>

// max of two values of any type
template <typename T>
T const& max(T const& a, T const& b)
{
    return a < b ? b : a;
}
```

Slide 18

```
// max of two pointers
template <typename T>
T* const& max(T* const& a, T* const& b)
{
    return *a < *b ? b : a;
}

// max of two C-strings
char const* const& max(char const* const& a,
                       char const* const& b)
{
    return std::strcmp(a,b) < 0 ? b : a;
}
```

Slide 19

```
int main()
{
    int a = 7;
    int b = 42;
    ::max(a, b);    // max for two ints

    std::string s = "hey";
    std::string t = "you";
    ::max(s, t);    // max for two std::strings

    int* p1 = &b;
    int* p2 = &a;
    ::max(p1, p2);  // max for two pointers
```

Slide 20

```
    char const* s1 = "David";
    char const* s2 = "Nico";
    ::max(s1, s2);  // max for two C-strings

    return 0;
}
```

Be careful to pass call parameters as `const references`. Else temporary local values might be created which cannot be returned from a function by a reference.

Slide 21

Class Templates

```
#include <vector>
#include <stdexcept>

template <typename T>
class Stack {
  public:
    void push(T const& elem);
    void pop();
    T top() const;
    bool empty() const;
```

Slide 22

```
  private:
    std::vector<T> elems;
};
```

Class Templates . . . *(continued)*

Slide 23

```
template <typename T>
void Stack<T>::push(T const& elem)
{
  elems.push_back(elem);
}
```

Class Templates . . . *(continued)*

Slide 24

```
template <typename T>
void Stack<T>::pop()
{
  if (elems.empty()) {
    throw std::out_of_range
             ("Stack<T>::pop(): empty stack");
  }
  elems.pop_back(elem);
}
```

Slide 25

Class Templates . . . *(continued)*

```
template <typename T>
T Stack<T>::top() const
{
  if (elems.empty()) {
    throw std::out_of_range
              ("Stack<T>::top(): empty stack");
  }
  elems.back(elem);
}
```

Slide 26

Class Templates . . . *(continued)*

```
template <typename T>
bool Stack<T>::empty() const
{
  return elems.empty();
}
```

Slide 27

Use of Class Template Stack

```
#include <iostream>
#include <string>
#include <cstdlib>
#include "stack.h"

int main()
{
  try {
    Stack<int>         intStack;
    Stack<std::string> stringStack;
```

Slide 28

```
    intStack.push(7);
    std::cout << intScack.top() << std::endl;

    stringStack.push("hello");
    std::cout << stringStack.top() << std::endl;
    stringStack.pop();
    stringStack.pop();
  }
```

Slide 29

```
  catch (std::exception const& ex) {
    std::cerr << "Exception: "
              << ex.what() << std::endl;
    return EXIT_FAILURE;
  }
  return 0;
}
```

Slide 30

Use of Class Template `Stack` ...

```
void foo(Stack<int> const& s)
{
  Stack<int> istack[10];
  ...
}
```

Slide 31

Use of Class Template Stack ...

```
typedef Stack<int> IntStack;

void foo(IntStack const& s)
{
  IntStack istack[10];
  ...
}
```

Slide 32

Use of Class Template Stack ...

```
Stack<float*>        floatPtrStack;
Stack<Stack<int> > intStackStack;
```

Notice! There must be a space between the two right angle brackets, `Stack<Stack<int> >`

Else it is interpreted as the stream extraction operator.

Specializations of Class Templates

Slide 33

```
#include <deque>
#include <string>
#include <stdexcept>
#include "stack.h"

template<>
class Stack<std::string>
{
  public:
    void push(std::string const& elem);
```

Slide 34

```
    void pop();
    std::string top() const;
    bool empty() const;

  private:
    std::deque<std::string> elems;
};
```

Slide 35

Specializations of Class Templates ...

```
void
Stack<std::string>::push(std::string const& elem)
{
  elems.push_back(elem);
}
```

Slide 36

Specializations of Class Templates ...

```
void Stack<std::string>::pop()
{
  if (elems.empty()) {
    throw std::out_of_range
      ("Stack<std::string::pop(): empty stack");
  }
  elems.pop_back();
}
```

Slide 37

Specializations of Class Templates ...

```
std::string Stack<std::string>::top() const
{
  if (elems.empty()) {
    throw std::out_of_range
      ("Stack<std::string::top(): empty stack");
  }
  return elems.back();
}
```

Slide 38

Specializations of Class Templates ...

```
bool Stack<std::string>::empty() const
{
  return elems.empty();
}
```

Slide 39

Partial specialization

In the previous example the template `Stack<>` was fully specialized. That meant, no template parameters was free to choose by the user.

Templates may be partially specialized. This means that some template parameters are still free, and must be chosen by the user.

An example of a partial specialization follows below.

Slide 40

Partial specialization . . . *(continued)*

Given some user defined template class

```
template <typename T1, typename T2>
class MyClass {
    ...
};
```

with two template parameters `T1` and `T2`, the class can be partially specialized, such that the both template parameters have the same type.

Slide 41

Partial specialization . . . *(continued)*

```
// partial specialization:
// both template parameters have the same type
template <typename T>
class MyClass<T, T> {
    ...
};
```

Slide 42

Partial specialization . . . *(continued)*

```
// partial specialization:
// the second template parameters is of type int
template <typename T>
class MyClass<T, int> {
    ...
};
```

Slide 43

Partial specialization ... *(continued)*

```
// partial specialization:
// both template parameters are pointer types
template <typename T1, typename T2>
class MyClass<T1*, T2*> {
    ...
};
```

Slide 44

Use of the Partially Specialized Templates

```
int main()
{
  MyClass<int, float>   mif; // MyClass<T1, T2>
  MyClass<float, float> mff; // MyClass<T, T>
  MyClass<float, int>   mfi; // MyClass<T, int>
  MyClass<int*, float*> mp;  // MyClass<T1*, T2*>

  return 0;
}
```

Slide 45

Use of the Partially Specialized Templates

If more partial specialization matches equally well, the declaration is ambiguous.

```
MyClass<int,int> m;
  // ERROR: matches MyClass<T,T>
  //        and MyClass<T,int>

MyClass<int*,int*> m;
  // ERROR: matches MyClass<T,T>
  //          and MyClass<T1*,T2*>
```

Slide 46

Use of the Partially Specialized Templates

To resolve the last ambiguity, an additional partial specialization for pointers of the same type might be provided.

```
template <typename T>
MyClass<T*, T*> {
    ...
};
```

Slide 47

Default Template Arguments

Template parameters may have default values.

```
#include <vector>
#include <stdexcept>
template
  <typename T, typename CONT = std::vector<T> >
class Stack {
  public:
  ... member functions: push, pop, top, empty
  private:
    CONT elems;
};
```

Slide 48

Default Template Arguments . . .

The class template `Stack` has two template parameters, so the member functions must be implemented taking this into account.

As an example, the member function `push(T const& elem)` must be implemented as

```
template <typename T, typename CONT>
void Stack<T, CONT>::push(T const& elem)
{
  elems.push_back(elem);
}
```

Slide 49

Default Template Arguments ...

The class template `Stack` can be used in two ways:

```
int main()
{
  ...
  Stack<int> intStack;
    // uses a vector<int> as its container
  Stack<int, std::deque<int> >;
    // uses a deque<int> as its container
  ...
}
```

The operations on both `Stack`s are identical.

Slide 50

Nontype Class Template Parameters

Until now all template parameters have been types.

Template parameters can also be just plain constants. A fixed-size `Stack` may be implementes as follows:

```
#include <stdexcept>
template <typename T, int MAXSIZE>
class Stack {
    ... members: push, pop, top, empty, full
  private:
    T elems[MAXSIZE];
    int numElems;    // next free entry
};
```

Slide 51

Nontype Class Template Parameters ...

For this implementation the class template `Stack` needs a default constructor

```
template <typename T, int MAXSIZE>
Stack<T,MAXSIZE>::Stack() : numElems(0)

{
  for (int i = 0; i < MAXSIZE; ++i) {
    this->elems[i] = T();
  }
}
```

Slide 52

Nontype Class Template Parameters ...

```
template <typename T, int MAXSIZE>
void Stack<T,MAXSIZE>::push(T const& elem)
{
  if (numElems == MAXSIZE) {
    throw std::out_of_range
          ("Stack<>::push(): stack is full");
  }
  elems[numElems++] = elem;
}
```

Slide 53

Nontype Class Template Parameters ...

```
template <typename T, int MAXSIZE>
void Stack<T,MAXSIZE>::pop()
{
  if (numElems <= 0) {
    throw std::out_of_range
          ("Stack<>::pop(): empty stack")
  }
  --numElems;
}
```

Slide 54

Nontype Class Template Parameters ...

```
template <typename T, int MAXSIZE>
T Stack<T,MAXSIZE>::top()
{
  if (numElems <= 0) {
    throw std::out_of_range
          ("Stack<>::top(): empty stack")
  }
  return elems[--numElems];
}
```

Slide 55

Nontype Class Template Parameters ...

```
template <typename T, int MAXSIZE>
bool Stack<T,MAXSIZE>::empty() const
{
  return numElems == 0;
}
```

Slide 56

Nontype Class Template Parameters ...

```
template <typename T, int MAXSIZE>
bool Stack<T,MAXSIZE>::full() const
{
  return numElems == MAXSIZE;
}
```

Slide 57

Nontype Function Template Parameters

Function templates can also have nontype parameters. The function template

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

returns the sum of its argument `x` and the template parameter `VAL`.

Slide 58

Nontype Function Template Parameters

This function might be useful when used with an `STL` container, named `source`, if one wants to add a number to all the values of the container `source` and save the result in a container `dest`.

This must be done as follows:

```
std::transform(source.begin(), source.end(),
               dest.begin(),
               (int (*)(int const&))
                    addValue<int,5>);
```

Slide 59

Nontype Function Template Parameters

It is *not* possible to just write

```
std::transform(source.begin(), source.end(),
               dest.begin(),
               addValue<int,5>);
```

because all function templates are supposed to be overloaded.

Therefore, exactly one instance *must* be specified, as show previously.

Slide 60

Restrictions

There are a number of restrictions on nontype function template parameters:

- They *must* be *constant integral values.*
- Floating point numbers and class-type objects are *not allowed.*

Slide 61

Using Templates in Practice

When you use templates in your programs and you use the `g++` compiler then you should write a header file containing both the declaretion (the headers) and the definition (the implementation).

When you compile your programs, you should use the option `-frepo` to the `g++` compiler.

Slide 62

Tricky Basics

- Read this chapter and try to understand as much of as you can. Remember that this chapter describes some strange things, which should be clear later in the course (hopefully).
- Return to this chapter when you have template problems. Reread it and understand the strange things that were too weird the first time you read it.

Next, the focus will be on the keywords: `typename` and `template`.

Slide 63

Tricky Basics . . . keyword `typename`

Until now, you have seen the keyword `typename` used to specify template parameters for both function templates and class templates.

Sometimes, the keyword `typename` must also be specified at other places. An example is, when a declaration depends on a template parameter.

The reason is to tell the compiler that the following is a template, and make it interpret the angle brackets `<` and `>` correctly.

Slide 64

Tricky Basics . . . keyword `typename`

The *containers* (`vector, deque`) of the standard library `STL` contains type definitions of *iterators*

```
#include <iostream>
...
// print elements of an STL container
template <typename T>
void printcoll(T const& coll)
{
  typename T::const_iterator pos;
  typename T::iterator end(coll.end());
```

Slide 65

```
    for (pos = coll.begin(); pos != end; ++pos) {
      std::cout << *pos << ' ';
    }
    std::cout << std::endl;
  }
```

Or, if you want to have a local variable in the `for` statement

```
    for (typename T::const_iterator pos;
         pos != end; ++pos) {
      std::cout << *pos << ' ';
    }
```

Slide 66

Tricky Basics . . . keyword `template`

Until now, you have seen the keyword `template` used to specify a function template or a class template.

But template parameters can be templates themselves. In those cases, the keyword `template` must also be specified at other places.

Let's go to the bone, and show the final example with all bells and whistles (the book takes in smaller steps).

Containers in `STL` have two parameters, one of them is default. Therefore, this example will be more complicated.

Slide 67

Tricky Basics . . . keyword `template`

```
#include <deque>
#include <memory>
...
template <
  typename T,
  template <
    typename ELEM,
    typename ALLOC = std::allocator<ELEM>
  >
  class CONT = std::deque  // must be class here
>
```

Slide 68

```
class Stack {
  private:
    CONT<T> elems;
  public:
    ... member functions push, pop, top, empty
};
```

The `template template` parameter `CONT` is instantiated with only *one* parameter, because the second parameter is default, and we have chosen to use that!

The second parameter *must* be specified, and here it is *always* the default parameter.

Slide 69

Tricky Basics . . . keyword `template`

The implementation of the member function `push` is

```
template <typename T,
          template <typename ELEM,
                    typename ALLOC> class CONT
         >
void Stack<T,CONT>::push(T const& elem)
{
  elems.push_back(elem);
}
```