Templates in ANSI C++

Advanced Template Programming

Angelika Langer

Trainer/Consultant

`http://www.AngelikaLanger.com`

© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Agenda - Class Templates

- **Run-time vs. compile-time polymorphism**
- Template parameters and arguments
- Static members
- Template specialization
- Nested typedefs, `typename` , and traits

© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

A Means of Generalization

```
class Stack {
private:   int* v; int* p; int sz;
public:
 Stack(int s) { v = p = new int[sz=s]; }
 ~Stack() { delete[] v; }
 void push(int i) { *p++ = i; }
 int pop() { return *--p; }
 int size() const { return p-v; }
};
```


last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

A Means of Generalization

```
template <class T>
class Stack {
private:   T* v; T* p; int sz;
public:
 Stack(T s) { v = p = new T[sz=s]; }
 ~Stack() { delete[] v; }
 void push(T i) { *p++ = i; }
 T pop() { return *--p; }
 int size() const { return p-v; }
};
```


last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Run-Time Polymorphism

```
class Rotatable {
public:
  virtual void rotate(int) = 0;
};

class Ellipse : public Rotatable {
public:
  Ellipse(int x, int y)
  : Xradius(x), Yradius(y) { }
  virtual void rotate(int);
private:  int Xradius,Yradius;
};
```


last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Run-Time Polymorphism

```
void vertical_flip(Rotatable& d)
{ d.rotate(180); }


Ellipse ellipse(100,600);
vertical_flip(ellipse);

Rectangle rectangle(999,500);
vertical_flip(rectangle);
```


last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Compile-Time Polymorphism

```
class Ellipse{
public: Ellipse(int x, int y)
        : Xradius(x), Yradius(y) { }
        void rotate(int);
private:  int Xradius,Yradius;
};
class Rectangle {
public: Rectangle(int x, int y)
        : Xedge(x), Yedge(y)  { }
        void rotate(int);
private:  int Xedge, Yedge;
};
```


last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Compile-Time Polymorphism

```
template <class Rotatable>
void vertical_flip(Rotatable& d)
{ d.rotate(180); }


Ellipse ellipse(100,600);
vertical_flip(ellipse);

Rectangle rectangle(999,500);
vertical_flip(rectangle);
```


last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Run-Time vs. Compile-Time Dispatch

Inheritance-based polymorphism (OOP) can be replaced by templates (GP = generic programming).

last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Evaluation OOP vs. GP

Overhead:

- GP: code bloat & compile-time overhead
- OOP: run-time overhead

Conformance to an interface:

- GP: common names
- OOP: a common base class

Integration of built-in types:

- GP: overloaded operators <=> native operators
function objects <=> function pointers
- OOP: not possible

last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Defining a Class Template

```
template <class C> class String {
public:    String();
           String(const C*);
           String(const String&);
           C read(int i) const;
private:   struct Srep;
           Srep* rep;
};
```

C is the *template parameter* and stands for a type name.

String is a *class template*.

last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Using a Class Template

```
String<char> cs;
String<unsigned char> us;
String<wchar_t> ws;

class Jchar { /* Japanese character */ };
String<Jchar> js;
```

String<char> is the name of a class.

A class generated from a template is called a *template class*.

last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Template Instantiation

Generating a class from a class template is called *template instantiation*.

```
void f()
{ String<char> cs;
  cs = "Hello World!"
}
```

generates declaration of:

- classes `String<char>` and `String<char>::Srep`,
- `constructor`, `destructor`, and
- `assignment operator`.

last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Template Parameter

Template parameters can be:

- type parameters
- non-type parameters
- template parameters

last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Type and Non-Type Parameters

```
template <class T, int s>
class Buffer {
public:  Buffer() : sz(s) {}
private: T v[s];
         int sz;
};

Buffer<char,127> cbuf;
Buffer<Record,8> rbuf;
```


last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Non-Type Parameters

A non-type template parameter can be:

- an integral or enumeration type,
- the pointer or reference to an object or function with external linkage, or
- a pointer to member.

A non-type template parameter must not be:

- type `void`, or
- a floating type.

last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Arguments to Non-Type Parameters

A non-type template argument can be:

- a constant expression,
- the address of an object or function with external linkage, or
- a non-overloaded pointer to member.

A non-type template argument must not be:

- a string literal.

© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Arguments to Non-Type Parameters

```
template <int* p> class X;
int global_i;
X<&global_i>; // ok: external linkage

int a[10];
X<&a[2]> x; // error: address of array element

struct S {int m; static int n;} s;
X<&s.m> x; // error: addr of non-static member
X<&S::n> x; // ok: address of static member
```


© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Arguments to Non-Type Parameters

```
enum M {min=-10,max=100000};
template <M m> class X;
X<max> x;      // ok

template <double& d> class X;
X<3.1415> x;   // error, not an l-value
double d;
X<d> x;        // ok
```


last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Arguments to Non-Type Parameters

```
template <class Tt, char* p> class X;

X<int,"BMW"> x;     // error: string literal

char p[] = "Mercedes";
X<int,p> x;         // ok
```


last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Template Template Parameters

```
template < class K, class V
         , template<class T> class C >
class Map {
  C<K> key;
  C<V> value;
};

template <class T> class myArray { .... };

Map<int,string,myArray> phone_numbers;
Map<string,long,vector> prices;
```


© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Template Typedefs

- If your compiler does not support template template parameters, use a *template typedef*.
- Require of the container that is allows *rebinding*:

```
template < class T> class Container {
public:
  template <class U> struct rebind
  { typedef Container<U> other; };
  ...
};
```


© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Template Typedefs

Instead of template template parameter:

```
template < class K, class V
         , template<class T> class C >
class Map {
  C<K> key;
  C<V> value;
};
```

use template typedef:

```
template < class K, class V, class C >
class Map {
  C<K> key;
  typename C::rebind<V>::other value;
};
```


last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Member Templates

A class can have type members.

```
class ios_base {
public:
  class failure;
  ...
};
```

A class template can have type members that are themselves templates.

```
template <class T> class X {
public:
  template <class U> class Y;
};
```


last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Template Typedefs

An example from the standard library:

```
template <class T> class allocator {
public:
  template <class U> struct rebind
  { typedef allocator<U> other; };
};
```

i.e. **`allocator::rebind<aType>::other`**

is a typedef for

`allocator<aType>`

© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Template Typedefs

```
template <class T, class Allocator>
class List {
private:
  class Node;
  typedef typename A::rebind<Node>::other
     NodeAllocator; // alias for Allocator<Node>
  ...
};
```


© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Parameterization vs. Type

Objects of the same type are distinguished by their object state's value.

If you add a property you can:

- add it to the object state, or
- add new types.

Example: a *Command* class

New properties: *redoable* and *undoable*

last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Object Parameterization

Extend the object state by adding flags for *redoable* and *undoable*.

```
class Command {
private: bool redoable, undoable;
public:  Command(..., bool redo, bool undo);
         void redo() { if(redoable) // ...
                       else // do nothing }
         void undo() { if(undoable) // ...
                       else // do nothing }
};
```


last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Object Parameterization

Advantage:

— easy, no deep thought required

Drawback:

— "fat" classes with questionable semantics

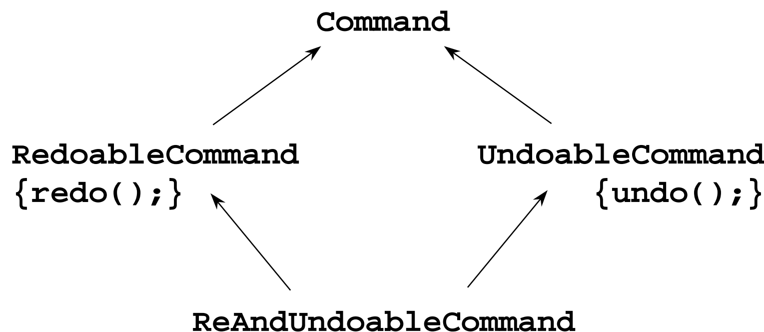
— hard to maintain

— number of combinations cannot be mastered; consistency checks for flag combinations needed

last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Expressing Properties as Types

Add new types *redoableCommand* and *undoableCommand.*



last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Lab: Time Values with Precision

Implement a class representing time values that have a precision, such as years, days, or seconds.

Provide means for calculation the difference between time values.

```
Ptime now;
Ptime Xmas(/* ctor args */);
cout << "It's still " << diff(Xmas,now) <<
"days 'til Xmas!\n";
```


© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Lab: Time Values with Precision

1st Idea: Add precision to the object state.

```
Ptime now(seconds);
Ptime Xmas(/* ctor args */,days);
cout << diff(Xmas,now);
```

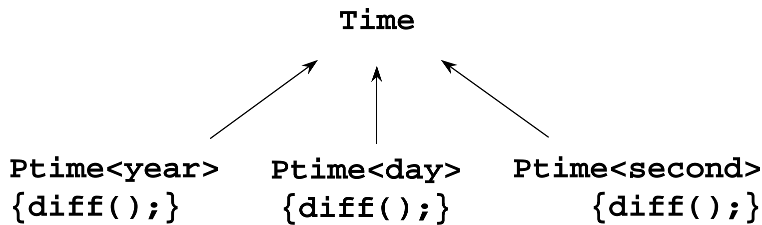
What is the difference between a time value in seconds and one in days?

Problems with operations that take two or more time values with different precision.

© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

The Derived Classes

Express different precision in terms of different types.



last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

The `Time` Base Class

```
class Time {
public:
 Time() : _t(time(NULL)) {}
 Time(long sec) : _t(sec) {}

protected:
 const time_t _t;
};
```


last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

The `Time` Base Class

```
class Time {
public:
 ...
 int getSeconds()
 { return localtime(&_t)->tm_sec; }

 /* and in the same way:  int getMinutes(); etc. */

 string toString()
 { return string(asctime(localtime(&_t))); }
};
```


last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

The Derived Class Template

```
template<long p>
class PTime : public Time {
public:
     typedef long TUnit;

     PTime<p>() {}
     PTime<p>(long sec) : Time(sec) {}

     TUnit diff(const PTime<p>& rhs)
     { return ((_t/p) - (rhs._t/p)); }
};
```


last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

For convenience ...

```
const long second = 1;
const long minute = 60 * second;
//  ... and so on ...

void main() {
 PTime<minute> pt1;
 PTime<minute> pt2(time(NULL) - 2*hour);

 cout << pt1.toString();
 cout << pt2.toString();

 cout << "diff is: " << pt1.diff(pt2) << endl;
}
```


© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Class Template Specialization

- *general class template:*
```
template<class Key, class Value>
class directory;
```
- *partially specialized class template:*
```
template<class Value>
class directory<long,Value>;
```
- *fully specialized class template:*
```
template<>
class directory<long,string>;
```


© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Code Bloat Reduction

```
template <class T>
class Vector {  // general vector type
 T * v;
 int sz;
public:
 Vector();
 Vector(int);

 T& elem(int i) { return v[i]; }
 T& operator[](int i);

 void swap(Vector&);
 // …
};
```


last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Code Bloat Reduction

Instantiations of the `Vector` template:

```
Vector<int>    vi;
Vector<Shape*> vps;
Vector<string> vs;
Vector<char*>  vpc;
Vector<Node*>  vpn;
```

For each instantiation the code of the member functions is replicated (=> code bloat).

last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Code Bloat Reduction

Many instantiations are for pointer types (to preserve run-time polymorphic behavior).

Specialization can be used for reducing the code bloat: vectors of pointers can share a single implementation.

=> Provide a version of the `Vector` template for pointer types, i.e. a specialization.

last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Code Bloat Reduction

Define a specialization for `void*` pointers:

```
template<> class Vector<void*> {
 void** p;
 // ...
 void*& operator[](int I);
};
```

Complete specialization; used for

```
Vector<void*> vpv;
```


last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Code Bloat Reduction

Implement a specialization for pointers types:

```
template<class T> class Vector<T*>
: private Vector<void*> {
public:
 typedef Vector<void*> Base;

 Vector() : Base() {}
 explicit Vector(int i) : Base(i) {}

 // ...
};
```


last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Code Bloat Reduction

```
template<class T> class Vector<T*>
: private Vector<void*> {
public:
 T*& elem(int i)
 { return static_cast<T*&>(Base::elem(i)); }
 T*& operator[](int i)
 { return
  static_cast<T*&>(Base::operator[](i)); }
 // ...
};
```


last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Code Bloat Reduction

Partial specialization; used for

```
Vector<Shape*> vps;  // T is Shape
Vector<int**> vppi;  // T is int*
```

`Vector<T*>` is an *interface* to `Vector<void*>`.

No overhead due to inline expansion.

General technique: Minimize code bloat by maximizing the amount of shared code.

last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Retrieving Template Arguments

```
template<class T, int s> class Buffer {
public:
 T& operator[](int i)  { return buf[i];}
};

Buffer<int,11> b; // range [-5,+5]
 for (int i = -5; i<= 5; i++)  b[i+5] = i;

Ranged<Buffer<int,11> > rb(-5,5);
 for (int i = -5; i<= 5; i++)  rb[i] = i;
```


last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Retrieving Template Arguments

```
template<class T, int s> class Buffer {
public:
 T& operator[](int i)  { return buf[i];}
};

template<class Cont> class Ranged {
public:
 value_type& operator[](int i)
 { return c[i-lb]; }
};
```

What is the return type of operator[] ???

last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Retrieving Template Arguments

We might try this:

```
template<class T, class Cont>
class Ranged
{ public:  T& operator[](int i); };
```

// redundant:

```
Ranged<int, Buffer<int,11> > rb(-5,5);
```

// error-prone:

```
Ranged<string, Buffer<int,11> > rb(-5,5);
```


last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Retrieving Template Arguments

There is a relationship between a template class and its template arguments.

It can be expressed by defining:

- nested typedefs for type arguments and
- static constants for non-type arguments.

© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Nested Typedefs and Static Constants

```
template<class T, int s> class Buffer {
public:
 typedef T val_t;
 static const int siz = s;

 Buffer() {};

 val_t& operator[](int i)
 { return buf[i];}
private:
  val_t buf[siz];
};
```


© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Nested Typedefs and Static Constants

```
template<class Cont> class Ranged {
public:
 class Error {};
 Ranged(int low, int high)
 {if (high-low+1 > Cont::siz) throw Error;
  else                        lb=low;
 }
 typename Cont::val_t& operator[](int i);

private:
 int lb; // lower bound
 Cont c;
};
```


last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Nested Typedefs and Static Constants

```
template<class Cont> class Ranged {
public:
 typedef Cont cont_t;
 typedef typename Cont::val_t val_t;
 static const int siz = Cont::siz;
 // ...
};
```


last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Typename

Type arguments can have members that are either data members, function members, or nested types.

```
class X {
 static int d;                // data member
 int f(int);                  // function member
 typedef int siz_t;           // type member
};
```


last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Typename

```
template <class T> class Adaptor {
 int f(T t)
 { s = T::siz_t(0); // is siz_t a function?
   i = T::f(0);
   x = T::d;
 }
};
```

Unless otherwise stated, an identifer is assumed to refer to something that is *not* a type.

last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Typename

```
template <class T> class Adaptor {
 int f(T t)
 { s = typename T::siz_t(0);
   i = T::f(0);
   x = T::d;
 }
};
```


last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Typename

You can also use `typename` as an alternative to `class` in a template declaration:

```
template <typename T> class X;
```

instead of

```
template <class T> class X;
```


last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Traits

```
char, wchar_t    // built-in character types

struct Jchar {  // Japanese characters
 wchar_t c;
 encoding e;
};
struct Achar {  // Arabic characters
 char c;
 orientation o;
};
```


last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Traits

```
template <class Char>
class String {
public:
 // ctors, dtor, ...
 int length() const;
 int compare (const String&);
 // ...
private:
 Char* p;
};
```


last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Traits

```
String::length { // for type char
  for (int i=0; p[i] != '\0'; i++);
  return i;
};

String::length { // for any character type
  for (int i=0;
       p[i] != end_of_string_char; i++);
  return i;
};
```


© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Traits

```
String::compare(const String& s) {
  // ...
  if ( p[i] < s.p[i] ) // code comparison
                // e.g. 'Z' (0x5A) < 'a' (0x61) in ASCII
};
// in general:
String::compare(const String& s) {
  // ...
  if ( less_than(p[i],s.p[i]) )
};
```


© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Traits

```
template <class Char>
struct charTraits {
 static const Char EOS;
 static bool less(Char, Char);
};

template <class Char, class Traits>
class String;
```


last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Traits

```
template <class Char, class Traits>
String::length {
 for (int i=0;p[i] != Traits::EOS; i++);
 return i;
};
template <class Char, class Traits>
String::compare(const String& s) {
  // ...
  if ( Traits::less(p[i],s.p[i]) )
};
```


last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Traits

```
struct Jchar {  // Japanese characters
 wchar_t c;
 encoding e;
 static const Jchar EOS;
 static bool less(Jchar j1, Jchar j2)
 { return (j1.c < j2.c); }
};
const Jchar::EOS = Jchar(0);
```


last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Traits

```
template <class Char>
struct charTraits {
 static const Char EOS = Char::EOS;
 static bool less(Char c1, Char c2)
 { return Char::less(c1,c2); }
};

String<Jchar,charTraits<Jchar> > // OK
String<Achar,charTraits<Achar> > // OK
String<char,charTraits<char> >   // ???
```


last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Traits

```
template <>
struct charTraits<char> {
 static const char EOS = '\0';
 static bool less(char c1, char c2)
 { return (c1 < c2); }
};
struct charTraits_Dictionary {
 static const char EOS = '\0';
 static bool less(char c1, char c2)
 { // order according to dictionary sorting rules }
};
```


last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Default Template Arguments

```
template <class Char
 , class Traits = charTraits<Char> >
class String;

String<char> // is same as
String<char,charTraits<char> >

// non-default traits:
String<char,charTraits_Dictionary>
```


last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Traits

... are a means to express the relationship between a template argument and associated static functionality or values.

character type <=> end-of-string character (value)

character type <=> comparison (functionality)

last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Agenda - Function Templates

- **Function template arguments**
- Function template overloading
- Member templates

last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Declaring and Using a Function Template

```
template <class T> void sort(vector<T>&);

void f(vector<int>& vi, vector<string>& vs)
{ sort(vi);     // sort(vector<int>& v);
  sort(vs);     // sort(vector<string>& v);
}
```

sort is a *function template*.

Template arguments are deduced from the function arguments.

last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Function Template Arguments

Type and non-type template arguments are deduced from a call, provided the function argument list uniquely identifies the set of template arguments.

```
template <class T, int i>
T lookup(Buffer<T,i>& b,const char* p);

class Record { ... };

int f(Buffer<Record,128>& buf, const char* p)
{ return lookup(buf,p); }
```

`T` is reduced to `Record` and `i` is `128`.

last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Deducing Class Template Arguments

Use function template argument deduction for class template argument deduction:

```
template<class T> class X { ... };

template<class T>
inline X<T> make_X(const T& t)
{ return (X<T>(t)); }

make_X(5);          // creates X<int>
make_X("Mary");     // creates X<const char*>
```


© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Class Template Arguments

Why is this useful?

```
template<class T>
inline void foo(const X<T>& t);
// we can say:
foo(make_X("Mary"));
// instead of:
foo(X<const char*>("Mary"));
```

Typically used when unnamed objects are passed to a function (function objects, manipulators, pairs).

© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Explicit Argument Specification

Function template arguments can be explicitly specified:

```
template<class T> class X { ... };

template<class T>
inline X<T> make_X(const T& t)
{ return (X<T>(t)); }

make_X<double>(5);          // creates X<double>
make_X<string>("Mary");     // creates X<string>
```


© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Explicit Specification needed if ...

a template argument cannot be deduced; typically, if the template argument is part of the return type of a function template:

```
template <class T>
T* create(); // make a T and return a pointer to it

int* p create<int>();
```


© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Argument Deduction Rules

A type argument T can be deduced if template parameter and argument have the following form:

- —T, const T, volatile T,
- —T*, T&,
- —T[const_expr],
- —class_template_name<T>,
- —type (*)(T), T (*)(args)

© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Conversions on the parameter type

The top-level const- or volatile-qualifier of a template type parameter is ignored for type deduction.

```
template <class T> void foo(const T);
int i; foo(i); => T=int
```

If the parameter is a reference type, the type referred to is used for type deduction.

```
template <class T> void foo(T&);
int i; foo(i); => T=int
```


© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Conversions on the argument type

If the argument is an array type, the pointer type (produced by array-to-pointer standard conversion) is used for type deduction.

```
template <class T> void foo(T*);
int a[5]; foo(a); => T=int
```

If the argument is a function type, the pointer type (produced by function-to-pointer standard conversion) is used for type deduction.

last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Argument Deduction Rules

class_template_name<T>

```
template <class T> void foo(X<T>);
X<int> x; foo(x); => T=int
```

Note:

```
template <class T> void foo(X<T>::Y);
X<int>::Y y; foo(y); => T cannot be deduced
```


last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Special Rule for Inheritance

If the type parameter is a reference or a pointer to class of the form class_template_name<arguments>, the argument can be a derived class.

```
template <class T> class B;
template <class T> class D1 : public B<T>;
class D2 : public B<T>;

template <class T> void foo(B<T>&);
D1<int> d1; foo(d1); => T=int, i.e. calls foo(B<int>&)
D2 d2; foo(d2); => T=int, i.e. calls foo(B<int>&)
```


last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Not covered ...

- rules for non-type template arguments
- rules for template template arguments
- deduction from pointer-to-member
- special rules for references and pointers (qualification conversions)

last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Scope of Template Parameters

The scope of a template parameter extends from its point of definition until the end of its template.

A template parameter can be used in the declaration of subsequent template parameters.

```
template <class T, T* P> class X {};
```

A template parameter can be used in the declaration of subsequent function parameters.

```
template <class T> void foo(T,T::V);
```

`Vector<int> v;  foo(v,5);` => T=Vector<int> and 5 must be convertible to Vector<int>::V

last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Function Template Specialization

```
template <class T>
void sort(vector<T>&)
{// Shell sort (Knuth)
 const size_t n=v.size();
 for ( int gap=n/2; 0<gap; gap/=2 )
   for ( int i=gap; i<n; i++ )
     for ( int j=i-gap; 0<=j, j-=gap )
       if ( v[j+gap] < v[j] )
         swap(v[j], v[j+gap]);
}
```


last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Function Template Specialization

sort<char*> does not sort a **Vector<char*>** correctly, because it compares the addresses of two strings, not the character sequences pointed to.

Replace *v[j+gap] < v[j]* by *less(v[j+gap], v[j])* and specialize **less()** for **char***:

```
template<> bool less<const char*>
 (const char* a, const char* b)
{ return (strcmp(a,b) < 0); }
```


last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Function Template Specialization

... is useful for

- more efficient alternatives to a general algorithm for a certain template argument, e.g. swap(), and
- for "irregular types" that lead to undesired results, e.g. less(); often true for pointers and arrays.

last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Function Overloading

```
void compare(string, string);
void compare(string, const char*);
void compare(string,date);

string s;
compare(s,"Xmas");
compare(s,date(12,24,98));
```


© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Function Template Overloading

```
template<class T> T sqrt(T);
template<class T>
complex<T> sqrt(complex<T>);
double sqrt(double);

void f(complex<double> z)
{ sqrt(2);      // sqrt<int>(int)
  sqrt(2.0);    // sqrt(double)
  sqrt(z);      // sqrt<complex<double>>(complex<double>)
}
```


© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Function Template Overload Resolution

For each function template find the specialization that is best for the set of function arguments.

Then apply usual function overload resolution rules to these specializations and all ordinary functions.

last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Member Templates

```
class IntStore {
public:
  // ...
 void add(const int* s, int n)
 { for (int i=0; i<n; i++)
       a[cur++] = s[i];
 }
private:
 int a[10];
 int cur;
};
```

```
IntStore buf;
short sarr[] = {10,20};
buf.add(sarr,2);  // error:
// cannot convert array of shorts to pointer to int
```


last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Member Templates

A class can have members that are templates.

```
class IntStore {
public:
 template <class T>
 void add(const T* s, int n)
 { for (int i=0; i<n; i++)
      a[cur++] = s[i]; // assign T to int
 }
};
```

```
IntStore buf;
short sarr[] = {10,20};
buf.add(sarr,2);  // IntStore::add<short>
```


last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Member Templates

A class template can have members that are themselves templates.

```
template <class Int>
class Store {
public:
 template <class Cont>
 void add(const Cont& s, int n=s.len())
 { for (int i=0; i<n; i++)
     a[cur++] = s[i];// assign containter element to Int
 }
};
```


last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Member Templates

```
Store<long> buf;

long arr[];
buf.add(arr,3);  // Store<long>::add<long*>

Store<int> buf2;
buf.add(buf2);   // Store<long>::add<Store<int> >
```


last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Member Templates

Member templates cannot be virtual.

```
class Shape {
 template<class T>
 virtual bool intersect(const T&)
  const = 0; // error: virtual template
};
```

The linker would have to add a new entry to the virtual function table each time `intersect()` is called with a new argument type.

last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Generic Conversions

```
template <class Scalar>
class complex {
 Scalar re, im;
public:
 template <class T>
 complex(const complex<T>& c)
 : re(c.re), im (c.im) {}
 // ...
};       // requirement: a Scalar can be initialized by a T
```


last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Generic Conversions

```
complex<float> cf(0,0);
complex<double> cd = cf;  // ok: converts float
                          // to double

class Quad {  // no conversion to int
};
complex<Quad> cq;
complex<int> ci = cq;  // error: cannot convert
                       // Quad to int
```


last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Generic Conversions

Constructor templates allow "generic conversions":
We can create a complex<X> from a complex<Y>, if data members of X can be created from data members of Y.

Note:

Constructor templates are never used for generating a copy constructor. You must explicitly define it if you need one.

© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Templates and Inheritance

```
class Shape { /*...*/ };
class Circle : public Shape { /*...*/ };
class Triangle : public Shape { /*...*/ };

void f(Vector<Shape*>& v)
{ ... v[i]->draw(); ... }
Vector<Circle*> v;
f(v);// error: type mismatch
     // Vector<Circle*> cannot be converted to Vector<Shape*>
```


© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Template Conversions

```
template <class T>
class Ptr {  // smart pointer
 T* _p;
public:
 Ptr(T*);
 template <class U>
 operator Ptr<U> ();  // convert Ptr<T> to Ptr<U>
 // ...
};
```


last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Template Conversions

```
Ptr<Circle> pc;
Ptr<Shape> ps = pc;  // should work,
  because Circle is derived from Shape
Ptr<Circle> pc2 = ps;  // should give error

Ptr<int> pi;
Ptr<void> pv = pi;  // should  work,
  because all pointer types can be converted to
  void*
```


last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Template Conversions

```
template <class T> class Ptr {
private: T* _p;
public:  Ptr(T* p) : _p(p) {}
         template <class U>
         operator Ptr<U> ()
         { return Ptr<U>(_p); }
};
```

The return statement will only compile if `_p` (which is of type `T*`) can be an argument to the constructor of `Ptr<U>`.

last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Contact Info

Angelika Langer

Training & Mentoring
Object-Oriented Software Development in C++ & Java

Munich, Germany

http: **//www.AngelikaLanger.com**

last update: 07.11.2005 ,21:59
© Copyright 1995-98 by Angelika Langer. All Rights Reserved.

Contact Info

© Copyright 1995-98 by Angelika Langer. All Rights Reserved.