Advanced Software Engineering with C++ Templates

Lecture 4: Separate Compilation and Templates III

Thomas Gschwind <thg at zurich dot ibm dot com>

Agenda

- Separate Compilation
  - Introduction (C++ versus Java)
  - Variables
  - Routines (Functions & Operators)
  - Types (Structures, Classes)
  - Makefiles
- Templates III
  - Design and Implementation (C++ versus Java versus C#)
  - “Dynamic” Static Algorithm Selection
  - Binders

Separate Compilation

- Why?
  - Having only one source file is unrealistic
  - Break the code up into its logical structure
  - Reduction of compile time
    - Only changed parts need to be recompiled
- How?
  - Use multiple source files
  - Need to know what information about functions and variables "used" from other files

Separate Compilation in Java

- Each Java file is compiled into a class file
- If a Java class invokes a method of another class, the compiler consults that other class file to
  - Determine whether the class provides the requested method
  - Determine whether a class file implements a given interface
  - etc.
  - Hence, the .class file contains the entire interface
- That's why in Java, the compiler needs the class path
- Finally, all class files are loaded by the Java Virtual Machine (and "linked")
- Java source code can be relatively well reconstructed from .class file (see Java Decompiler: jd, jad)

Some Java Trivia

- Let us write Hello World in Java
- In order to simplify the reconfiguration (e.g., translation)
  - Put all the constants into one Java file
  - Put the complex application code into another

```
public class Const {
  public static final String msg="Hello World!";
}
```

```
public class Cool {
  public static void main(String[] args) {
    System.out.println(Const.msg);
  }
}
```

Some Java Trivia

- Now compile both Java files and run Cool

```
public class Const {
  public static final String msg="Hello World!";
}
```

```
public class Cool {
  public static void main(String[] args) {
    System.out.println(Const.msg);
  }
}
```

- Change the msg in Const.java, recompile Const.java, and run Cool

Some Java Trivia

- Now compile both Java files and run Cool

```
public class Const {
  public static final String msg="Hello World!";
}
```

```
public class Cool {
  public static void main(String[] args) {
    System.out.println(Const.msg);
  }
}
```

- Change the msg in Const.java, recompile Const.java, and run Cool
- Cool still prints the old message! Why?
  - javac inlines constants
  - And according to the Java specification that's legal

Separate Compilation in C++

- By convention, each source file is compiled into an object file
- Object files provide minimum necessary to execute the code
- Object files do not provide enough information for the compiler to identify
  - Functions provided by another compilation unit
  - Layout of a user-defined type
- Object files are not used during the compilation
  - C++ and C use “header” files to store the interfaces of the object file
  - These files need to be supplied by the developer
- In C/C++, we have the include path instead

Header Files

- C++ uses so-called header files for separate compilation
- The header file can be viewed as the object file's interface
- Hence, header files are another encapsulation mechanism
  - They describe the interface(s) provided by the object files
  - Describe everything that should be exported to other compilation units
- What goes into the header file?
  - "Everything" that should be exported (i.e., used in other files)
  - "Nothing" that causes the compiler to immediately generate code
  - Except, for a small number of exceptions

Header Files: Prevent Multiple Inclusion

- Header files need protection from being included multiple times
- Otherwise, this may cause compile errors

if VARS_H_ is not defined process the following lines

define _VARS_H

end the last open #if… section

This pattern ensures that header files won't be included multiple times

Process the file "vars.h"

vars.h

```
#ifndef VARS_H_
#define VARS_H_
extern int my_dumb_global_variable;
extern const double e;
const double pi=3.141596;
#endif
```

vars.cc

```
#include "vars.h"

int my_dumb_global_variable=17;
const double e=2.718281;
```

Header Files: Variables

- Variables
  - Declaration goes into the header
    (if variable is to be accessed elsewhere)
  - Definition goes into the implementation file (allocates memory)
- Constant Variables
  - Declaration goes into the header
    (if variable is to be accessed elsewhere)
  - Definition goes either into the header or the implementation file
  - If in the header, definition may be allocated multiple times
    - No problems for individual values (constant and small)
    - Be careful with large constants such as large constant arrays

Header Files: Variables Example

vars.h

```
#ifndef VARS_H_
#define VARS_H_
extern int my_dumb_global_variable;
const double pi=3.141596;
extern const int[] primes;
#endif
```

Use extern to declare a variable to be defined elsewhere. No memory will be allocated for the variable

Constants may be defined in the header or declared like other variables.

vars.cc

```
#include "vars.h"

int my_dumb_global_variable=17;
const int primes[]={2, 3, 5,…, 1234567891};
```

Include the header (for consistency checking)

Variables are defined in the implementation file. Do not repeat constants defined in the header.

Header Files: Functions

- Functions
  - The declaration of the function goes into the header
  - Definition goes into the implementation file
- Inline Functions
  - If they are to be inlined in the corresponding implementation file only, treat them like functions
  - If they are to be inlined globally (typical), declaration and definition go into the header file
  - Necessar for the compiler to know the implementation to be use instead of the function call

Header Files: Functions Example

util.h

```
#ifndef UTIL_H_
#define UTIL_H_
inline void swap(int &a, int &b) {
  int c=a; a=b; b=c;
}
extern int gcf(int a, int b);
inline int lcm(int a, int b) {
  return (a/gcf(a,b))*b;
}
#endif
```

Inline functions are declared and defined in the header file.

Use extern to declare a function. Extern for function declarations is optional. If there is no function body, it cannot be a definition.

util.cc

```
#include "util.h"
int gcf(int a, int b) {
  if (a<b) swap(a,b);
  while (b!=0) { a=a-b; if (a<b) swap(a,b); }
  return a;
}
```

Functions are defined in the implementation file. Do not repeat inline functions defined in the header.

Header Files: Types (`typedef`, `struct`, `class`)

- The type declaration and definition go into the header
  - The layout of the type needs to be known to the compiler in all compilation units that need to allocate the type
- For members of a type the same rules as for functions apply
  - Member declarations into the header
  - Member definitions (unless it should be inlined) into the implementation file
  - All members of a class even if they should not be visible outside the compilation unit need to be declared as part of the type definition

Header Files: `class` Example

fraction.h

```
class fraction {
  int c; int d;
public:
  fraction(int cntr=0, int denom=1) : c(cntr), d(denom) { /*void*/ }
  fraction operator*(const fraction &b);
  fraction operator/(fraction b) {
    swap(b.c, b.d); return (*this)*b;
  }
};
```

The complete layout of a type (public, protected, private members) go into the header file.

`operator/` is an implicitly inline member, inline functions go into the header file.

fraction.cc

```
#include "fraction.h"
#include "util.h"

fraction::fraction operator*(const fraction &b) {
  fraction r;
  int f1=gcf(this->c,b.d), f2=gcf(b.c,this->d);
  r.c=(this->c/f1)*(b.c/f2);
  r.d=(this->d/f2)*(b.d/f1);
  return r; }
```

Header Files: Templates

- The type declaration and definition go into the header
  - The code for a template is generated when it is parameterized
  - Hence, the compiler needs the full code to instantiate the template
- If the template is only to be parameterized with a small set of types
  - Can treat template functions and template classes like normal functions and classes
  - Need to instantiate the class in an implementation file that has access to the full template definitions
    (for instance, `template class pvector<string>;`)

main program

- Include header files
  - System header files first allows to find compile errors in header more easily
  - Own header files first allows to find missing definitions in header more easily
- Compile each file
- Put dependencies into Makefile
  - If implementation file changes, it needs to be recompiled
  - If a header file changes, layouts of types may have changed, all files including it need to be recompiled

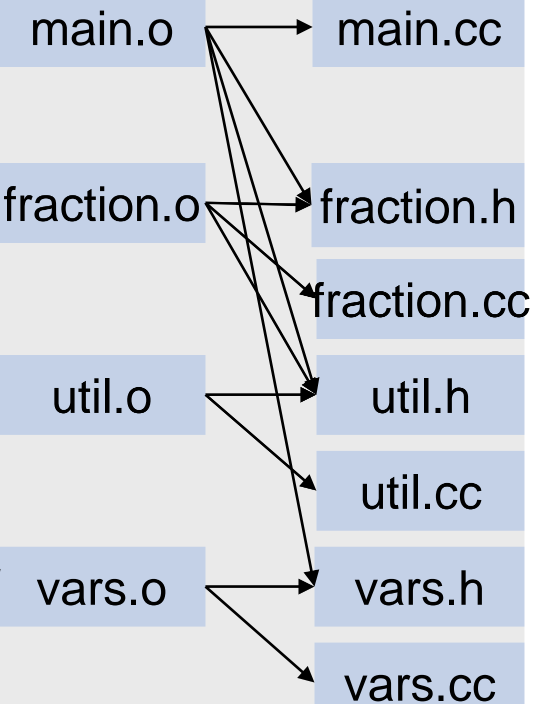
```
#include <stdlib.h>
#include <iostream>
#include "fraction.h"
#include "util.h"
#include "vars.h"

void main(int argc, char *argv[]) {
  int arg=atoi(argv[1]);
  cout << arg << "^2*pi="
       << arg*arg*pi << endl;
  cout << "e=" << e << endl;

  cout << gcf(atoi(argv[1]),
              atoi(argv[2])) << endl;
  cout << lcm(atoi(argv[1]),
              atoi(argv[2])) << endl;

  … // use of fraction data type
}
```

Makefile

Makefile

Link the final executable
(could also use ldd)

```
all: main

main: main.o fraction.o util.o vars.o
	g++ -o main main.o fraction.o util.o vars.o

main.o: main.cc fraction.h util.h vars.h
	g++ -c main.cc

fraction.o: fraction.cc fraction.h util.h
	g++ -c fraction.cc

util.o: util.cc util.h
	g++ -c util.cc

vars.o: vars.cc vars.h
	g++ -c vars.cc
```



Useful Build Tools

- gcc: not only compiles and links files
  - Also analyzes sources and generates dependencies between them (Checkout the –M… options)
  - Also allows to analyze java files for the dependencies between them
- nm: list symbols in an object file or program
  - Defined, undefined, which section, etc.
- ldd, otool –L: list libraries needed by an object file
  - Simply list the shared libraries the object file is dependent on

Makefiles (made easy)

- As mentioned before, gcc allows to generate source dependencies
- This Makefile can be used as a generic starter for your Makefile

```
CFLAGS=...
CFLAGS+=-Wall -Wextra -Werror

OBJS=main.o                                     # "main" file
OBJS+=fraction.o ...                            # others…

all: main

clean:
        rm -f main *.o

distclean: clean
        rm -f .depend/*.d
        rm -f *~
...
```

Makefiles (cont'd)

```
...

-include $(addprefix .depend/,$(OBJS:.o=.d))

%.o: %.c
	gcc $(CFLAGS) -c -o $@ $*.c
	@gcc -MM $(CFLAGS) -c $*.c >.depend/$*.d

main: $(OBJS)
	ld $(LDFLAGS) -o $@ $(OBJS)
```

Agenda

- Separate Compilation
  - Introduction (C++ versus Java)
  - Variables
  - Routines (Functions & Operators)
  - Types (Structures, Classes)
  - Makefiles
- Templates III
  - Design and Implementation (C++ versus Java versus C#)
  - "Dynamic" Static Algorithm Selection
  - Binders

C++ Templates – Implementation

- C++ creates a new artifact for each new use of a template (almost all modern C++ compilers do some optimization to avoid unnecessary code duplication)
- Similar to advanced C macros

C++ Templates – Implementation (cont'd)

- The code for a template is generated when it is parameterized
- The compiler needs the full code to instantiate the template

```
template<typename T>
T min(T a, T b) {
  return a<b?a:b;
}
const double pi=3.141596;
void f() {              // create implementation for
  min(2.718282,1.0);   // min<double>
  min('a','z');        // min<char>
  min(1,26);           // min<int>
  min(pi,2.718282);    // … already created
  min<char>('a',26);   // … already created
  min(2.718282,1.0);   // … already created
}
```

C++ Templates – Consequences

- C++ creates a new artifact for each new use of a template
  - Uses "more" memory
  - Templates are typically defined in the header file
    (Otherwise, the compiler cannot instantiate them when they are used)
  - Can be used in combination with built-in types
  - Can be used in combination with non-typenames
    (e.g., `int`s, `char`s, …)
  - Better optimization since each template instantiation can be optimized for the corresponding template argument
  - Longer compile times due to higher optimization potential

C++ Templates – Question

- What if we have multiple source files instantiating the same template?
  - file1.cc instantiating vector<int>
    file2.cc instantiating vector<int>
  - Will the code for vector<int> be generated twice once for each object file?
  - Will the executables be double the size when main.cc uses both of them?

C++ Templates – Question

- What if we have multiple source files instantiating the same template?
  - file1.cc instantiating vector<int>
    file2.cc instantiating vector<int>
  - Will the code for vector<int> be generated twice once for each object file?
    Yes!
  - Will the executables be double the size when main.cc uses both of them?
    No!
  - Why? The corresponding code of the templates is put into a COMDAT section of the object file. The linker only uses the first such section with the same name.

Java Generics – Implementation

- Java Generics are implemented using type erasure
  - Template parameters are internally replaced with their type bound (Object by default)
  - The compiler inserts casts as necessary

Source Code	"Generated" Code

`public class Iterator<E> {`
`    E next();`
`    boolean hasNext();`
`}`
 	
`Iterator<String> iter=…;`
`while (iter.hasNext()) {`
`  String s=iter.next();`
`  …`
`}`
	`public class Iterator {`
	`    Object next();`
	`    boolean hasNext();`
	`}`
	
	`Iterator iter=…;`
	`while (iter.hasNext()) {`
	`  String s=(String)`
	`    iter.next();`
	`  …`
	`}`

Java Generics – Consequences

- Java keeps only one copy of the class
  - Java Generics “cannot” be used with primitive types
  - Need to be encapsulated in the corresponding wrapper class
  - Java does this since quite some time automatic BUT
    generating wrappers requires memory, stresses the garbage collector
- Allows forward and backward compatibility with existing code

```
public String oops(Integer x) {
  List<String> ys = new LinkedList<String>();
  List xs = ys;
  xs.add(x);                       // compile-time unchecked warning
  return ys.iterator().next(); // run-time error
}
```

Java Generics – Consequences (cont'd)

- Java keeps only one copy of the class
  - Uses "more" memory because many dynamic casts need to be inserted
  - Programs can be unsafe although they look safe
    (but the compiler will warn about it)
  - Allows forward and backward compatibility with existing code
  - Cannot create elements of generic type E
    - No `new E` or `new E[]`
    - Especially the latter can be annoying, need to use `Object[]` and a lot of `(E)` casts which are unsafe (see previous slide)
  - JIT cannot optimize for different parameterizations

C# Generics – Implementation

- C# thought about Generics from the start
  - Unlike the Java VM, the CLR is already generic
  - Generics can be parameterized with primitive types
  - CLR JIT can optimize code for different parameterizations

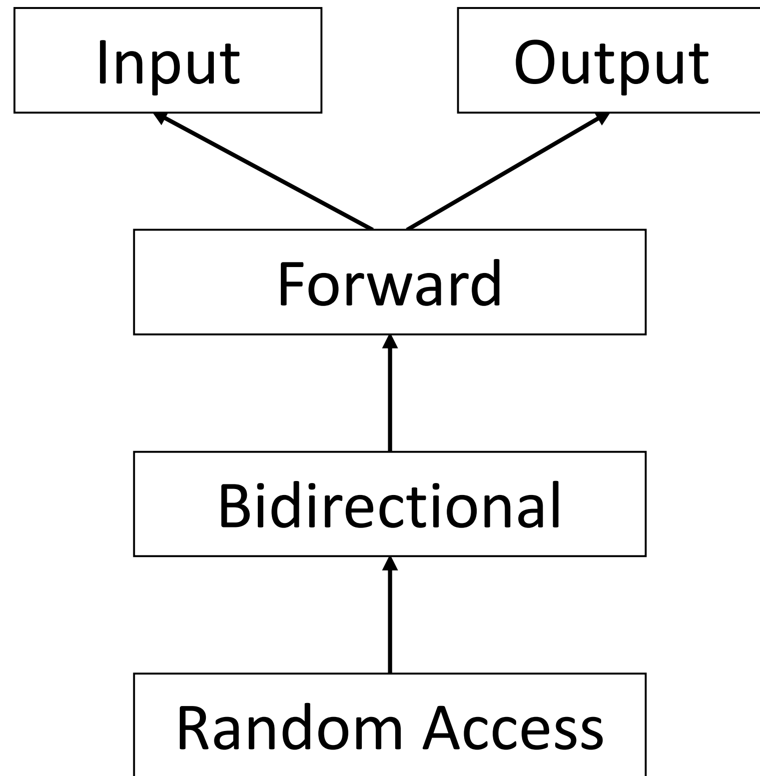
C# Generics – Consequences

- Templates are stored as such in the byte-code
- Can be used for built-in types
- Can use arrays of type T
- Cannot be used in combination with non-typenames (e.g., `int`s, `char`s, …)
- Better optimization since the template can be optimized for each specific template argument
- Uses "more" memory
- Is somewhere between C++ and Java

Different Types of Iterators

Category	output	input	forward	bi-directional	random-access
Abbrev.	Out	In	For	Bi	Ran
Read		=*p	=*p	=*p	=*p
Access		->	->	->	-> []
Write	*p=		*p=	*p=	*p=
Iteration	++	++	++	++ --	++ + += -- - -=
Compare		== !=	== !=	== !=	== < <= != > >=

Types of Iterators (cont‘d)




```
struct input_iterator_tag {};
struct output_iterator_tag {};

struct forward_iterator_tag :
  public input_iterator_tag {};

struct bidirectional_iterator_tag :
  public forward_iterator_tag {};

struct random_access_iterator_tag :
  public bidirectional_iterator_tag {};
```

Iterator Tags?

- Algorithm selection
- Saves typing overhead
- Simulation of bound genericity

> Decide which algorithms you want; parameterize them so that they work for a variety of suitable types and data structures.
>
> Bjarne Stroustrup

Why Iterator Tags? (cont'd)

iterator

```
template <class In> iterator_traits<In>::difference_type
__distance(In first,In last, input_iterator_tag dummy) {
  iterator_traits<In>::difference_type n = 0;
  while(first!=last) { ++first; ++n; }
  return n;
}
template <class Ran> iterator_traits<Ran>::difference_type
__distance(Ran first,Ran last, random_access_iterator_tag dummy) {
  return last-first;
}
template <class I> inline
iterator_traits<I>::difference_type distance(I first,I last) {
  typedef typename iterator_traits<I>::iterator_category cat;
  return __distance(first,last, cat());
}
```

iterator_traits

- Provide a generic iterator description that covers standard iterators and pointers

iterator

```
template <class Iter> struct iterator_traits<Iter> {
  typedef typename Iter::iterator_category iterator_category;
  typedef typename Iter::value_type value_type;
  typedef typename Iter::difference_type difference_type;
  typedef typename Iter::pointer pointer;
  typedef typename Iter::reference reference;
};
template <class T> struct iterator_traits<T*> {
  typedef random_access_iterator_tag iterator_category;
  typedef T value_type;
  typedef ptrdiff_t difference_type;
  typedef T* pointer;
  typedef T& reference;
};
```

Binders - Motivation

- Want to find an element in a container that fulfills a criterion
- For instance, first element in the container that is greater then a given value
- In Java
  - If the container provides this fine (unlikely)
  - If not, we start implementing our own routine

Locating an Element

- The straight forward solution is to use find_if with a helper function

```
int greater_than_17(int x) {
  return x>17;
}

void foo(vector<int> v) {
  vector<int>::iterator b=r.begin(), e=r.end(), i;
  i=find_if(b, e, greater_than_17);
  …
}
```

- Having to write that helper function is bothersome
- In C++11, a more readable solution would be a lambda function
- Any ideas how to improve this without using C++11?

Function Objects

Predicates `<functional>`

- `equal_to`, `not_equal_to`
- `greater`, `greater_equal`, `less`, `less_equal`
- `logical_and`, `logical_or`
- `logical_not` (unary)

Arithmetic Operations `<functional>`

- `plus`, `minus`, `multiplies`, `divides`, `modulus`
- `negate` (unary)

Locating an Element

- We say we want that the greater than function is executed
- Problem greater than takes two arguments
- Solution, the bind2nd function binds one argument to a given value

```
void foo(vector<int> v) {
  vector<int>::iterator b=r.begin(), e=r.end(), i;
  i=find_if(b, e, bind2nd(greater<int>(),17));
  …
}
```

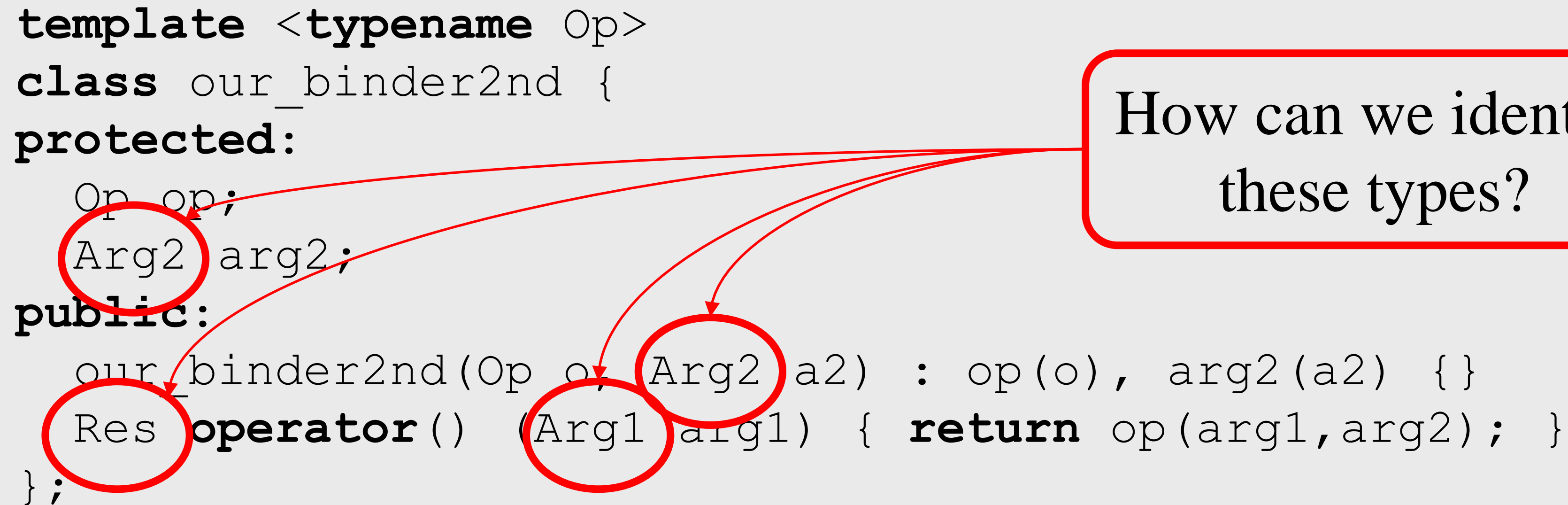
bind2nd(binop,arg2)

- Binds the second argument of a function
- If I have already a function less/2 one would not like to write another one less/1 for all possible arguments

- Implementation
  - Need to store a binary operation binop and the second argument arg2

  => We need a function object to implement this

our_binder2nd



- Solution
  - Function Object Bases
  - They provide standardized names for arguments, and return types

Function Object Bases

- Provide standardized names for arguments, and return types for function objects
- Use them religiously!

```
template <class Arg, class Res> struct unary_function {
  typedef Arg argument_type;
  typedef Res result_type;
};
template <class Arg, class Res> struct binary_function {
  typedef Arg first_argument_type;
  typedef Arg second_argument_type;
  typedef Res result_type;
};
```

bind2nd

```
template <class BinOp>
class binder2nd : public
  unary_function<BinOp::first_argument_type, BinOp::result_type>
{
protected:
  BinOp op;
  typename BinOp::second_argument_type arg2;
public:
  binder2nd(const BinOp &o,
            const typename BinOp::second_argument_type &a2)
    : op(o),arg2(a2) {}
  result_type operator() (const argument_type &arg1) {
    return op(arg1,arg2); }
};
template<class BinOp, class T> binder2nd<BinOp>
bind2nd(const BinOp &o, const T &v) {
  return binder2nd<BinOp>(o,v); }
```

Function Object Bases (Forts.)

> Using these bases [Function Object Bases] consistently the way the standard library does will save the programmer from discovering the hard way why they are useful (§18.4.4.1).

Binders, Adapters, Negaters

Binders `<functional>`

- `bind1st,bind2nd`

Adapters `<functional>`

- `mem_fun,mem_fun_ref,ptr_fun`

Negaters `<functional>`

- `not1,not2`

Summary

- Separate Compilation
  - Introduction (C++ versus Java)
  - Variables
  - Routines (Functions & Operators)
  - Types (Structures, Classes)
  - Makefiles
- Templates III
  - Design and Implementation (C++ versus Java versus C#)
  - “Dynamic” Static Algorithm Selection
  - Binders

Exercise 1: Merge STL Containers

- Implement a function that merges two STL containers. Think about whether you want to merge containers of different types (vector, list, set, map, etc.) or only containers of the same type.
- Look into the concepts we have learned so far, depending on how you want to implement this algorithm, they could be useful.
- Explain your design decisions, such as how to represent the containers (container, sequence, etc.), how to add the elements to the target container, and how to map fundamentally different containers (lists and maps).

Exercise 2: `find_if` in C++ and Java

- Implement a function findIf(Iterator iter, Matcher matcher) in Java that finds the first element in the sequence defined by iter. Matcher is an interface with a single method match return true if the element matches.
- Implement in a separate Java file a benchmark that executes the above on a Vector with several millions of elements.
- Implement the same benchmark in C++ with the C++ find_if method and, e.g., lambdas.
- Try to optimize BOTH (Java and C++) solutions. However, the matcher and the elements stored in the container shall be generic.

```
public static interface Matcher<E> {
    boolean match(E elem);
}
```

Exercise 3

- Which of these lines belong into the header file? Why?

```
char ch;
string s;
extern int error_number;
static double sq(double);
int count=1;
const double pi=3.2; // according to Indiana Pi Bill
struct fraction { int c; int d; };
char *prog[]={"echo","hello","world!",NULL};
extern "C" void c_swap(int *a, int *b);
double sqrt(double);
void swap(int &a, int &b) {  int c=a; a=b; b=c;  }
template<typename T> T add(T a, T b) { return a+b; }
namespace { int a; }
struct user;
```

Exercise 4: Separate Compilation

- Update all the exercises you implemented so far and split them into multiple compilation units (i.e., files) where you believe it makes sense.
- For instance the fraction class is a perfect candidate to put into a separate compilation unit.

Exercise 5

- What could be the purpose of the following code? Where could it be useful? Implement a program that demonstrates its use.

```
template <class BinOp, class Op1, class Op2>
class somefunction_t :
public unary_function<typename Op1::argument_type,
                      typename BinOp::result_type> {
protected:
  BinOp o; Op1 o1; Op2 o2;
public:
  somefunction_t(BinOp binop, Op1 op1, Op2 op2)
    : o(binop), o1(op1), o2(op2) {}
  typename BinOp::result_type
  operator()(const typename Op1::argument_type &x) {
    return o(o1(x),o2(x));
  }
};
```

Next Lecture

- Memory Management
- Classes: Fallacies and Pitfalls

Have fun solving the examples!

See you next week!