Research Centre Jülich

Writing Efficient Programs with C++

Jörg Striegnitz
Central Institute for Applied Mathematics
`J.Striegnitz@fz-juelich.de`

Version 1.18 - 03.01.02 / short version

General Overview

- Abstractions in C++
- The Abstraction Penalty
- How to beat it
- (Some) Existing Solutions

General Overview

- **Part I: Abstractions and their Cost**

  Short recapitulation of some C++ abstractions. Special emphasize on runtime performance and optimizations + a look under the hood of existing C++ implementations

- **Part II: Template Metaprogramming**

  Introduction of a special programming technique that allows for fascinating optimizations. It is especially this technique that distinguishes C++ from other languages (e.g. Fortran, C).

- **Part III: Advanced Techniques**

  ... be prepared :)

Important

In this course we are talking about

C++

Thus, C++ according to the

International Standard ISO/IEC 14882

and neither GNU C++, Microsoft Visual C++, Borland C++, or any other vendor specific C++ implementation !

I. Abstractions and their Cost

Jörg Striegnitz
Research Centre Jülich GmbH
j.striegnitz@fz-juelich.de

Overview

- *Classes*
- Polymorphism
  - Inheritance
  - Overloading
  - Conversions
  - Templates
- Further Topics
- References

I.1. Classes Introduction

What is a class ?

- user defined type
- provides methods for logical grouping of data and functionality
- provides methods for data-hiding and protection
- can be used to express hierarchical relationships between concepts (derived classes)

A class consists of

- **data memebers** are variables of arbitrary type (perhaps, classes)
- **member functions** are action/operations usually applied to data members (they define how a class behaves)
- **constructors** are called during creation of an object (initialize data members, request system resources)
- **destructor** is called during deletion of an object (free system resources)

An *instance* of a class is called **object**

I.1. Classes Definitions

Grady Booch: Object-Oriented Analysis and Design With Applications

An **object** has **state**, **behavior**, and **identity**; the structure and behavior of similar objects are defined in their common **class**; the terms instance and object are interchangeable.

James Rumbaugh: Object-Oriented Modeling and Design

An **object** is a **concept**, **abstraction**, or **thing**, with crisp boundaries and meaning for the problem at hand. Objects serve two purposes:

- they promote understanding of the real world and
- provide a practical basis for computer implementation.

Decomposition of a problem into objects depends on judgment and the nature of the problem. There is no one correct representation.

I.1. Classes Introduction - An Example

```
class doubleVec3 {
private:
   int Size;
   double* data;
protected:
   const double* getData() const {
      return data;
   }
public:
   int getSize() const {
      return size;
   }
};
```

- Class definitions are introduced by the keyword `class` and always end with a semicolon.
- Specifiers `public`, `private`, and `protected` are used to control access to members.
- `const` member functions do not change the object

I.1. Classes Lifecycle of an Object

The lifecycle of an object consists of five phases:

- **Allocation** of memory to hold data members - create space for object
  - **Construction** (done by ***constructors***): initialization of data members, potentially allocation of system resources (e.g. open a file)
    - **Usage**
  - **Destruction** (done by ***destructor***): free potentially allocated system resources (e.g. close a file)
- **De-Allocation** of memory to store data members

I.1. Classes Lifecycle of an Object - Constructor Types

There are three kinds of constructors

- **Default constructor** - may be called without supplying an argument. Thus, either doesn't take any, or all of them have a default value
- **Regular constructor** - takes at least one argument
- **Copy constructor** - used to create copy of object.
  (*Do not confuse with assignment operator !*)

- If there is no user-defined constructor, **default constructor** and **copy constructor** are automatically created by the compiler (*see next slide*)

- Constructors always have the name of the class

- Maybe placed in **`public`**, **`private`**, or **`protected`** section of class definition

I.1. Classes Lifecycle of an Object - Automatically generated Constructors

Automatically generated default constructor

- Empty member initializer list / empty function body, thus `A::A() {}`

Automatically generated copy constructor

- Has form `A::A(const A&)`, if each direct or virtual base class **`B`** of **`A`** has a copy constructor whose first parameter is of type **`const B&`** or **`const volatile B&`**, and for all the nonstatic data members of **`A`** that are of a class type **`M`** (or array thereof), each such class type has a copy constructor whose first parameter is of type **`const M&`** or **`const volatile M&`**, otherwise it has form `A::A(A&)`
- Performs memberwise copy of its subobjects

Implicitly declared constructors are **`public inline`** members of their class

Note 0: *Function Inlining*

`inline` functions help reducing the function call overhead

- push parameters to function on stack
- call the function (push instruction pointer to stack - jump to code of function)
- reserve stack space for local variables
- ... execute function ...
- clean stack space (local variables)
- return to caller
- clean stack (remove parameters)

The effect is to substitute each occurrence of the function call with the text of the body of the function.

```
inline int square(int i) {
  return i*i;
}
...
int i = square(3);
...
int j = square(10);
```

```
int i = 3*3;
...
int j = 10*10;
```

Note 0: *Function Inlining*

Note ! **`inline`** specifier is simply **a hint**, not a mandate to the C++ compiler !

- removes overhead
- provides better opportunities for further optimizations
- increases code size
- reduces efficiency if abused (e.g. code no longer fits cache)

More on inlining later ...

I.1. Classes Lifecycle of an Object - Automatically generated Constructors

• Take care when relying on automatically genereated constructors !

```
class Foo {
public:
   int myInt;
   int* myIntPtr;
};

Foo a;

void bar() {
   Foo b;
   if (b.myInt) {
   }
   ...
};
```

`myInt` and **`myIntPtr`** are not initialized to **`zero`** by automatically created default constructor !

Memory for global static variables is filled with zeroes

May fail or succeed - depending on the contents of the stack

I.1. Classes Lifecycle of an Object - Automatically generated Constructors - Augmenting I

```
class Foo { public: Foo(); ... };
class Bar { public: Bar(); Bar(int); ... };
class Zap { public: Zap(); ... };

class MyClass {
   int anInt;
public:
   Foo f;
   Bar b;
   Zap z;
};
```

Automatically synthesized constructor for **`MyClass`** invokes default constructors for each member object. **`anInt`** remains uninitialized

```
MyClass::MyClass(MyClass* this) {
   Foo::Foo(this->f);
   Bar::Bar(this->b);
   Zap::Zap(this->z);
}
```

I.1. Classes Lifecycle of an Object - Automatically generated Constructors - Augmenting II

What happens for the following case ? Do **f** and **z** remain uninitialized ?

```
class MyClass {
  int anInt;
public:
  MyClass() : b(1024) { anInt = 12; }
  Foo f;
  Bar b;
  Zap z;
};
```

No, our constructor gets augmented:

```
MyClass::MyClass(MyClass* this) {
  Foo::Foo(this->f);
  Bar::Bar(this->b,1024);
  Zap::Zap(this->z);
  anInt=12;
}
```

Notice, that the order of initializations corresponds to the order in which the members have been declared !

I.1. Classes Lifecycle of an Object - Destructors

- Destructors always have the name of the class prepended with ~
- Must be placed in **`public`** section of class definition (or **`protected`** section of abstract base class)
- Guaranteed to be invoked when an object gets destroyed
- If not defined, compiler generates one of form **`A::~A() {}`**

I.1. Classes Lifecycle of an Object - Constructor / Destructor Example

```
class doubleVec3 {
private:
  double* data;
public:
  doubleVec3() { // default constructor
    data=new double[3];
  }
  doubleVec3(const doubleVec3& rhs) { // copy constructor
    data = new double[3];
    data[0]=rhs.data[0]; data[1]=rhs.data[1]; data[2]=rhs.data[2];
  }
  doubleVec3( int val ) { // regular constructor
    data = new double[3];
    data[0]=val; data[1]=val; data{2]=val;
  }
  ~doubleVec3() { // destructor
    delete [] data;
  }
};
```

I.1. Classes Lifecycle of an Object - Time of Construction/Destruction

- A ***named automatic object*** is created each time its declaration is encountered in the execution of the program and destroyed each time the program exits the block in which it occurs.

Allocation: reserve space on stack
(e.g. decrement stack pointer by `d`'s size)

```
void foo() {
  doubleVec3 d;
}
```

Construction: call `d.doubleVec3()`

Destruction: call `d.~doubleVec3()`

De-Allocation: free stack space
(e.g. increment Stack pointer by `d`'s size)

- ***Passing arguments by value*** involves calling the copy constructor

Construction: call `f.doubleVec3(const doubleVec3&)`
with the argument passed to `bar`

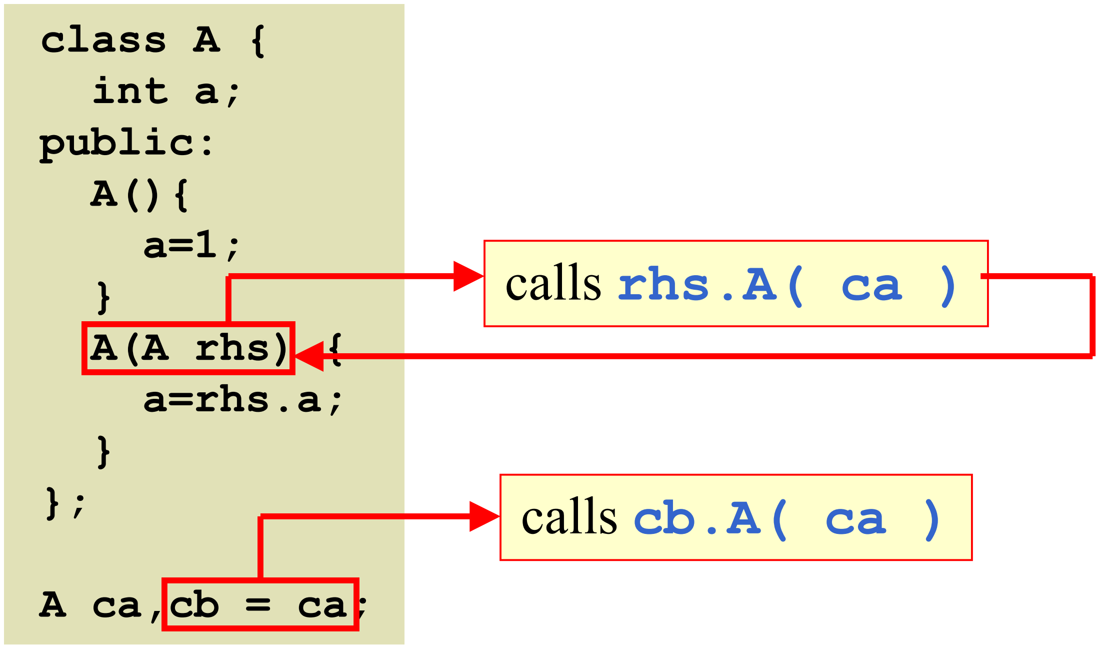
```
void bar(doubleVec3 f) {
}
```

Allocation, Destruction, De-Allocation:similar to previous case

Note 1: *Copy Constructor*

? **Why must copy constructors take their arguments by reference ?**

! **To avoid infinite recursion.**



```
class A {
   int a;
public:
   A(){
      a=1;
   }
   A(A rhs) {
      a=rhs.a;
   }
};

A ca,cb = ca;
```

Note 2: *Passing arguments to functions / class member functions*

! ***Passing arguments by reference may save runtime !***

- Passing arguments ***by reference*** avoids call to copy constructor

```
void bar(doubleVec3& f) {
}
```

- Pass arguments ***by const reference*** to avoid call to copy constructor and to avoid users from changing the associated object

```
void bar(const doubleVec3& f) {
  // f is read-only !
  doubleVec3 g = f; // o.k.
  f = g;            // illegal ! Yields compile-time error !
}
```

I.1. Classes Lifecycle of an Object - Time of Construction/Destruction 2

- A ***nonstatic member*** is created and destroyed with its hosting object.

```
class C {
  doubleVec3 d;
  ...
};

void foo() {
  C c;
}
```

Allocation: reserve stack space for `c` (includes `c.d`)

Construction: call `c.C()` - `d.doubleVec3()` gets called

Destruction: call `c.~C()` - `d.~doubleVec3()` get called

De-Allocation: free stack space

- A ***free store object*** is created using operator **`new`** and destroyed using operator **`delete`**

Allocation & Construction: allocate free store for `d` - call `d.doubleVec3(12)`

```
doubleVec3 *d = new doubleVec3( 12 );
...
delete d;
```

Destruction & De-Allocation: free heap space - call `d.~doubleVec3()`

I.1. Classes Lifecycle of an Object - Time of Construction/Destruction 3

- An ***array element*** is created and destroyed when the array of which it is an element is created and destroyed

Allocation & Construction: allocate free store for **100** elements of type `doubleVec3` and call `doubleVec3()` for each of them.

```
doubleVec3 *dArray = new doubleVec3[ 100 ];
...
delete [] dArray;
```

Destruction & De-Allocation: free heap space - call `~doubleVec3()` 100 times

remember to distinguish between operator `delete` and operator `delete[]`

- A ***global***, ***namespace***, or ***class static object*** is created at the start of the program and destroyed once at the termination of the program

I.1. Classes Lifecycle of an Object - Time of Construction/Destruction 4

- A ***local static object*** is created the first time its declaration is encountered and destroyed once at the termination of the program

Example:

```
void f(int i) {
   static doubleVec3 a;
   if (i != 0) {
      static doubleVec3 b;
      ...
   }
   ...
}
f(0);
...
f(1);
```

Allocation for a and b on start of program.

De-Allocation & Destruction of a and b at end of program.

f(0); ← **Construction** of **a** doesn't happen before this call.

f(1); ← **Construction** of **b** doesn't happen before this call - **a** isn't constructed again !

- A ***temporary object*** that is the result of an expression gets destroyed at the end of the full expression *(More on temporaries later)*.

I.1. Classes Lifecycle of an Object - Penalties

• Definition of an object does not only mean to reserve space but also to run some initialization code !

• Definition of an array or creating an array with operator **`new`** respectively, results in calling the initialization code for every item of this array.

• Leaving scope (e.g. function / for-loop) may mean to call de-initialization code for one or multiple object.

• Initialization overhead of super classes / class members maybe unknown

• Possibly doing initialization multiple times (redundant construction)

Note 3: *Redundant Construction*

! *A member initialization list may save runtime !*

```
class A {
   int a[100];
public:
   A()        { setA(255); }
   A(int b)  { setA(b);}
   setA( int b) {
     for (int i=0; i<100 ; i++)
       a[i] = b;
   }
};

class B {
   A a;
public:
   B( int j ) { a.setA(j); }
};
```

Expensive !

Better variant: use mem-initializer list

```
class B {
   A a;
public:
   B( int j ) : a(j) {}
};
```

Very bad: `a.setA` is called twice (once through default constructor) !

Note 4: *Member Initialization List (MIL) Pittfall*

! *Why does this program print “No” ?*

```
class S {
public:
   int i;
   int j;
   S(int v) : j(v), i(j) { }
};

S s(10);

int main() {
   if (s.i == s.j)
      cout << “Yes”;
   else
      cout << “No”;
   return 0;
};
```

The order of initializations in the MIL does not matter. Members are always initialized in the order they are defined !

```
S::S(S* this,int v) {
   this->i = this->j;
   this->j = v;
}
```

Note 5: *Lazy Construction*

! **Variables not necessarily need to be declared at the beginning of a function -** ***postpone variable declaration as long as possible***

```
class A {
  int a[100];
public:
  A()        { setA(255); }
  A(int b)  { setA(b);}
  setA( int b) {
    for (int i=0; i<100 ; i++)
      a[i] = b;
  }
};

void f(bool b,const A& z) {
  A x,y;
  if (b) {
    x = z;
    // do something with x
  };
  // do something with y
}
```

Point of construction may save runtime !

```
void f(bool b,constA& z) {
  if (b) {
    A x = z;
    // do something with 'x'
    // (possibly "return" !)
    // A::~A(x) destructor call
  };
  A y;
}
```

Note 6: *Explicit Initialization*

! ***If possible: prefer explicit initialization !***

```
class A {
  int a[100];
public:
  A()       { setA(255); }
  A(int b)  { setA(b);}
  setA( int b) {
    for (int i=0; i<100 ; i++)
      a[i] = b;
  }
};

void f(bool b,const A& z) {
  A x;
  // ...
  x = z;
  // ...
}
```

`A x;` → Default constructor gets called here

`x = z;` → **`operator=`** gets called !

```
void f(bool b,constA& z) {
  A x = z; // also possible A x(z);
}
```

`A x = z;` → **just invokes the copy constructor**

Note 7: *Anticipated Destruction*

! **Sometimes it may be reasonable to destroy objects earlier than the current scope ends -> open a new scope !**

```
class A {
  int* a;
public:
  A() { a = new int[100];
        setA(255);      }
  ~A() { delete [] a;}
  setA( int b) {
    for (int i=0; i<100 ; i++)
      a[i] = b;
  }
};

void f(bool b,const A& z) {
  A x,y;

  //... do something with x
  // x no longer needed
  //... do something with y

} // x and y get destroyed here
```

Note : Explicitly calling the destructor (e.g. **`x.~A()`**) may cause trouble (resource gets freed twice !)

Point of destruction may help to save space !

```
void f(bool b,constA& p) {
  { A x = p;
    // do something with 'x'
  } // x gets destroyed here !
  A y;
} // y gets destroyed here
```

Note 8: *Constructors and Caching*

! ***Be aware of the architecture of your system !***

```
class A {
   int a;
   int ar[8192];
   int c;
public:
   A() : a(1),c(2) {}
}
```

A standard-conforming constructor will lay out an object of type **`A`** in the declaration order ! **`a`** is separated by 8192 bytes from **`c`**

Will probably cause a cache miss !

```
class A {
   int a;
   int c;
   int ar[8192];
public:
   A() : a(1),c(2) {}
}
```

Reorder members to avoid possible cache miss

Order of data members may influence caching behavior !

Overview

- Classes
- *Polymorphism*
  – Inheritance
  – Overloading
  – Conversions
  – Templates
- Further Topics
- References

I.2. Polymorphism Definition

What is a polymorphism ?

- from Greek language: *having multiple forms*

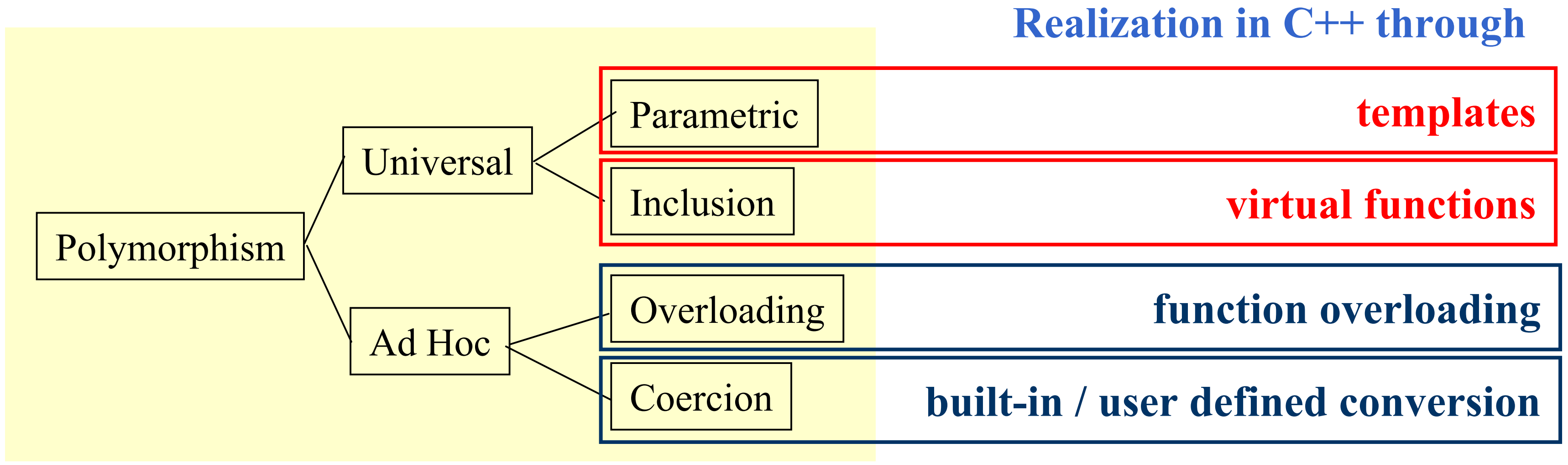
Strachey (1967)

Parametric polymorphism is obtained when a function works uniformly on a range of types; these types normally exhibit some common structure. **Ad-hoc polymorphism** is obtained when a function works, or appears to work, on several different types (which may not exhibit a common structure) and may behave in unrelated ways for each type.

Cardelli and Wegner (1985)

Refined Strachey's definition by adding **inclusion polymorphism** to model subtypes and subclasses (inheritance). Strachey's parametric polymorphism is divided into **parametric** and **inclusion polymorphism**, which are closely related, but separated to draw a clear distinction between the two forms, which are then joined as specializations of the new **universal polymorphism**.

I.2. Polymorphism Polymorphism in C++



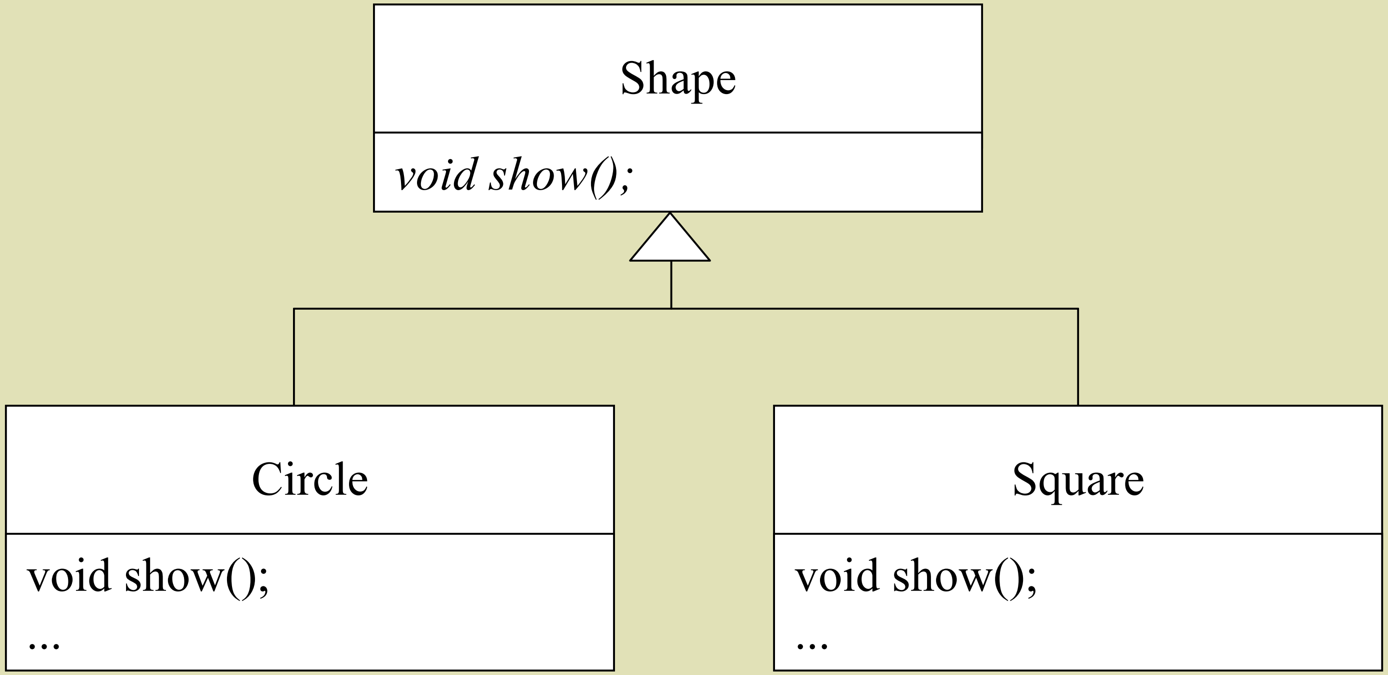
Polymorphism according to Cardelli & Wegner

In OO languages **inclusion polymorphism** is often called **subtype polymorphism**

Overview

- Classes
- *Polymorphism*
  - *Inheritance*
  - Overloading
  - Conversions
  - Templates
- Further Topics
- References

I.2.1. Subtype Polymorphism Virtual Functions



I.2.1. Subtype Polymorphism Virtual Functions

Example:

```
class Shape {
public:
  virtual void show() = 0;
};

class Circle : public Shape {
  double r_m;
public:
  void show(); // virtual !
  Circle(double r) : r_m(r) {}
};

class Square : public Shape {
  double r_l;
public:
  void show(); // virtual !
  Square(double l) : r_l(l) {}
};
```

```
Shape* Sptr = new Square(3);
Circle c(4);

Shape& Sref = c;
Shape  S;      // illegal !

Sptr->show(); // Square::show
Sref.show();  // Circle::show
S.show();     // illegal !!
```

If a virtual member function is **overridden** by a derived class and called through a *pointer* or *reference*, the type of the pointed to/ referenced object determines which function gets called.

I.2.1. Subtype Polymorphism Virtual Functions 2

- If a virtual member function is **overridden** by a derived class and called through a *pointer* or *reference*, the type of the *pointed to/referenced* object determines which function gets called.

- Type of *pointed to/referenced* object maybe unresolvable during compile-time

- **Therefore: dynamic binding** (or late binding) is a **runtime mechanism**

Redefining a virtual member function in a derived class is called **overriding**

Remark: Some people distinguish **overriding** and **augmenting**, whereas overriding means to completely substitute the implementation of the base class and augmenting refers to extending the implementation of the base class (by e.g. making a call to it).

Note 9: *Parameterized Inheritance*

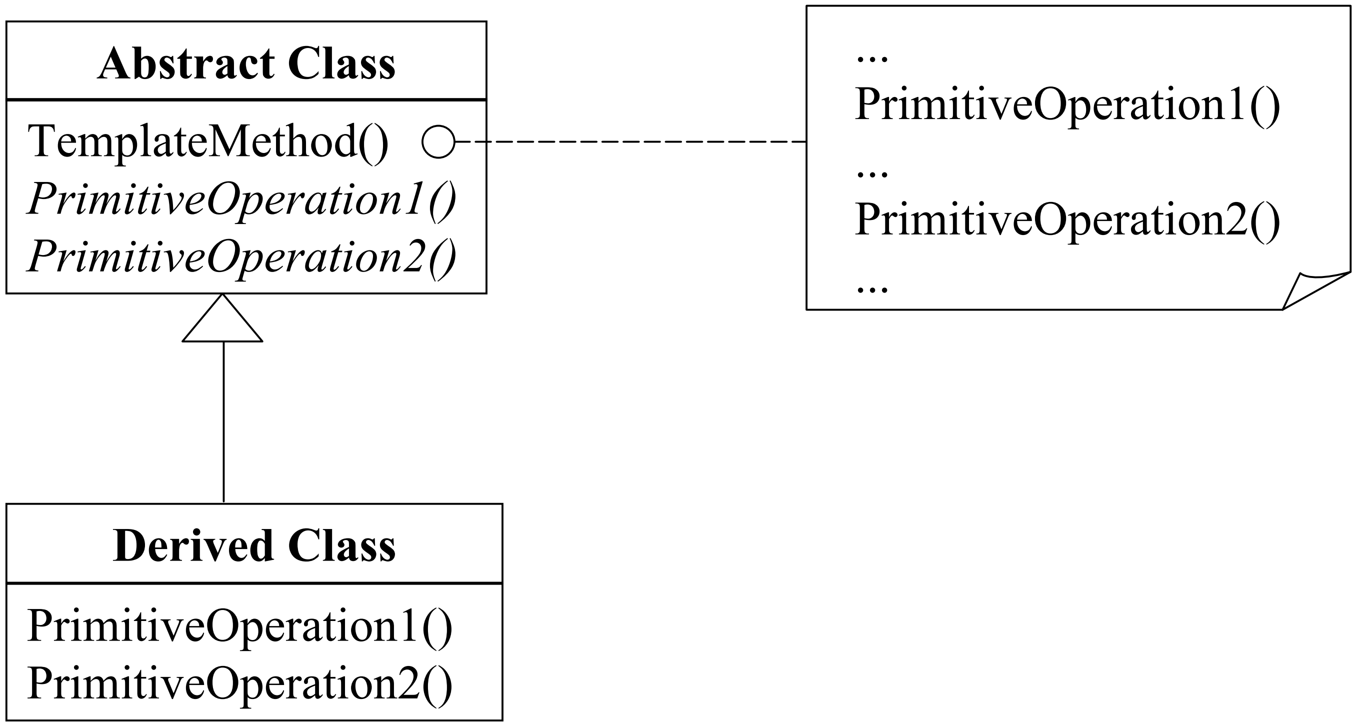
! **C++ supports parameterized inheritance / subtype polymorphism**

```
template <class SuperClass>
class DerivedClass : public SuperClass {
...
};
```

This technique often is used to implement static **wrappers** or **template methods**.
It also maybe used as an alternative to multiple inheritance for implementing **mixin**-based designs.

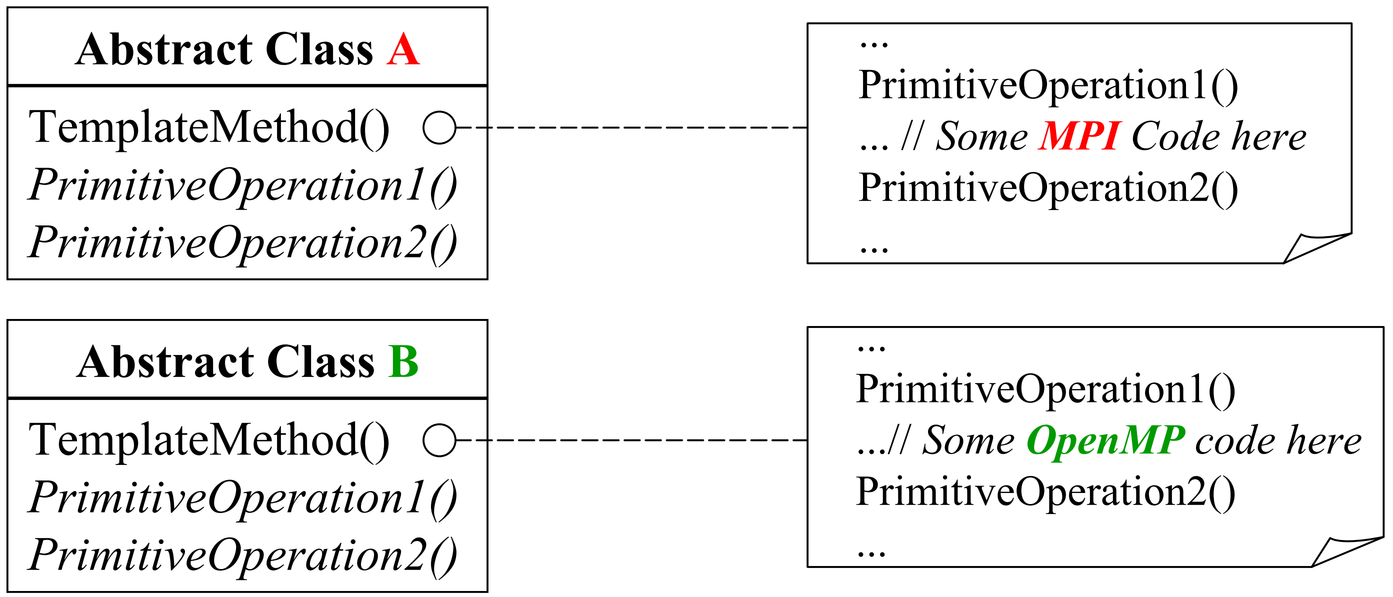
Note 9: *Parameterized Inheritance*

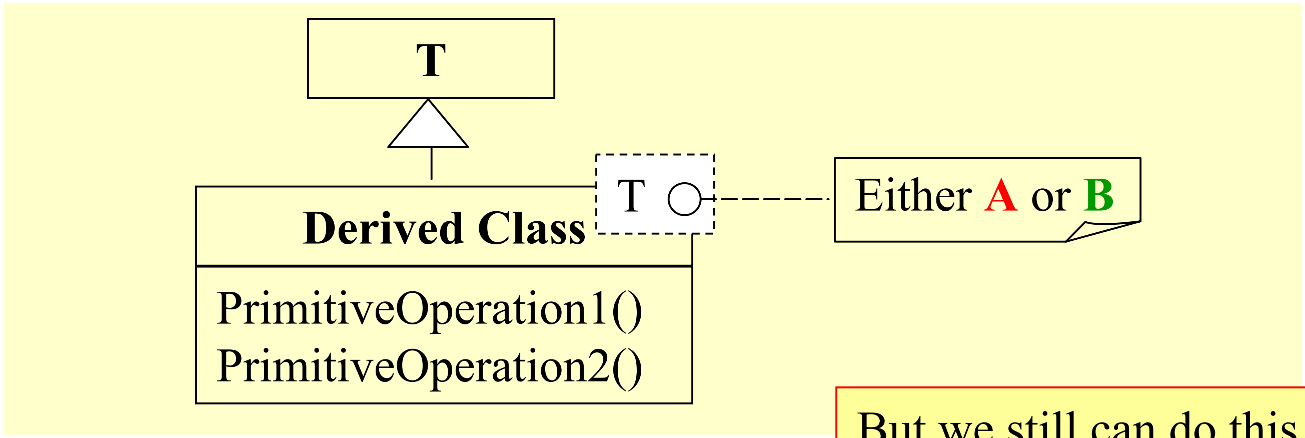
Example: Template Method



Note 9: *Parameterized Inheritance*

Example: Template Method - **Increased flexibility** through parametrized inheritance





But we still can do this better ...

Note 9: *Parameterized Inheritance - Pitfall*

! **Take care when using similar names for methods / functions !**

```
void afunc() {
  cout << "global afunc !" << endl;
}

class A {
protected:
  virtual void afunc() {
    cout << "A's afunc !" << endl;
  }
};

template <class SuperClass>
struct DerivedClass : public SuperClass {
  void tfunc() {
    afunc();
  }
...
};

DerivedClass<A> a;
a.tfunc();
```

Always calls global **`afunc`** !
(Relation to **`SuperClass`** is unclear !)

Prints "**`global afunc ! `**"

Note 9: *Parameterized Inheritance - Pitfall - Solution*

! **Take care when using similar names for methods / functions !**

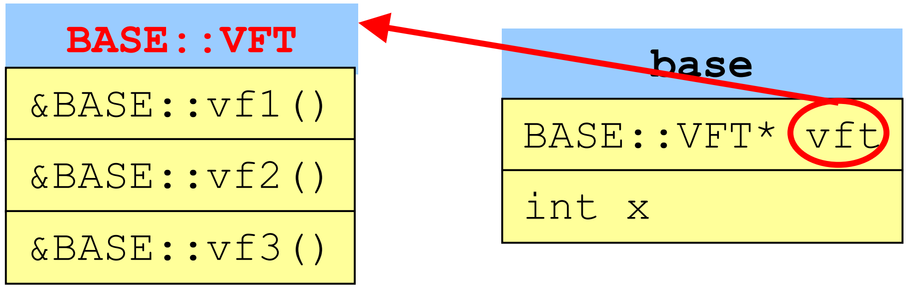
```
void afunc() {
  cout << "global afunc !" << endl;
}

class A {
protected:
  virtual void afunc() {
    cout << "A's afunc !" << endl;
  }
};

template <class SuperClass>
struct DerivedClass : public SuperClass {
  typedef DerviedClass<SuperClass> self;
  void tfunc() {
    self::afunc();
  }
...
};
```

Now relation is clear !

I.2.1. Subtype Polymorphism Method Dispatch - Basics

```
class BASE {
private:
  int x;
public:
  virtual void vf1();
  virtual void vf2();
  virtual void vf3();
};

BASE* base = new BASE;
```

Create **Virtual Function Table** (**VFT**) for every class with virtual members or inherited virtual members:



Object contains pointer to **VFT**

Call to `base->vf1()` is transformed into `base->vft[0](base)`

Remember the implicit parameter (`this`)

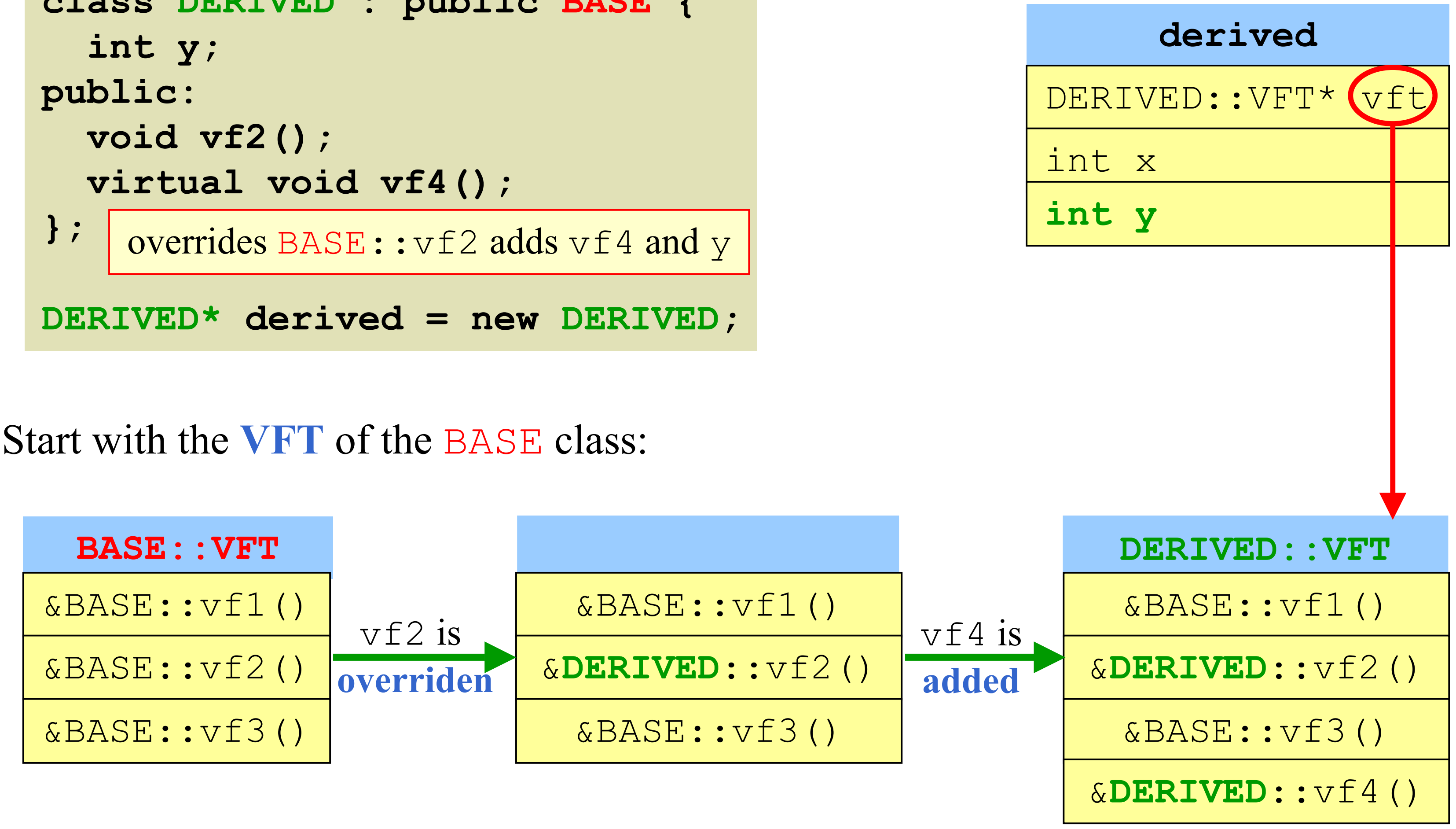
I.2.1. Subtype Polymorphism Method Dispatch - Single Inheritance

```
class DERIVED : public BASE {
  int y;
public:
  void vf2();
  virtual void vf4();
};
```

overrides BASE::vf2 adds vf4 and y

```
DERIVED* derived = new DERIVED;
```

Start with the VFT of the BASE class:



I.2.1. Subtype Polymorphism Method Dispatch - C-style type casting

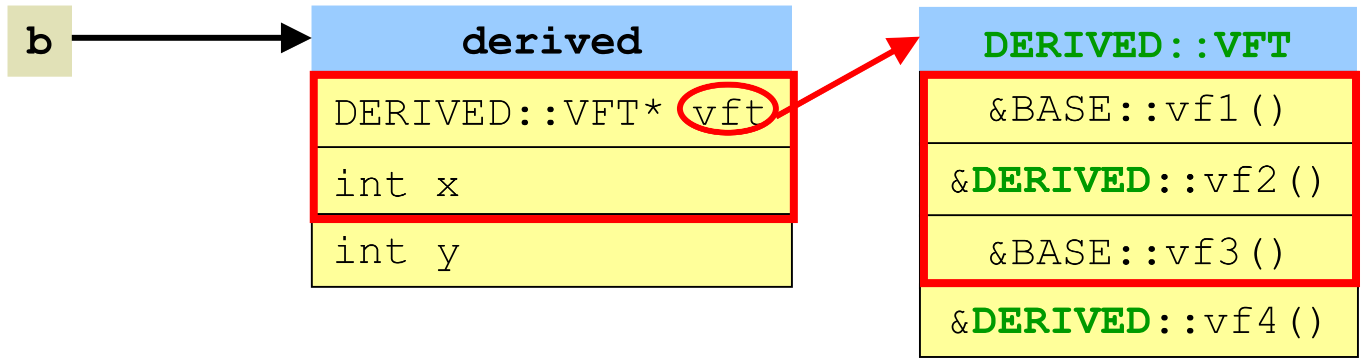
• Single inheritance utilizes **C-style casting**

C-style *type casting*

C-style type casting an object of type A into an object of type B means to handle the address space used by A as if it contains an object of type B.

`BASE* b=new DERIVED;`

Object of type **DERIVED** is *type casted* into object of type **BASE**



The `derived` object's memory layout is compatible with objects of type BASE.

points to **DERIVED::vf2**

Call to `b->vf2()` is transformed into `b->vft[1](base)`

I.2.1. Subtype Polymorphism Method Dispatch - Multiple Inheritance

- C-style casting does not work under the presence of **Multiple Inheritance** (**MI**)

```
class BASE1 {
public:
   BASE1();
   virtual ~BASE1();
   virtual void speakClearly();
   virtual BASE1* clone() const;
protected:
   float data_BASE1;
};
```

```
class BASE2 {
public:
   BASE2();
   virtual ~BASE2();
   virtual void mumble();
   virtual BASE2* clone() const;
protected:
   float data_BASE2;
};
```

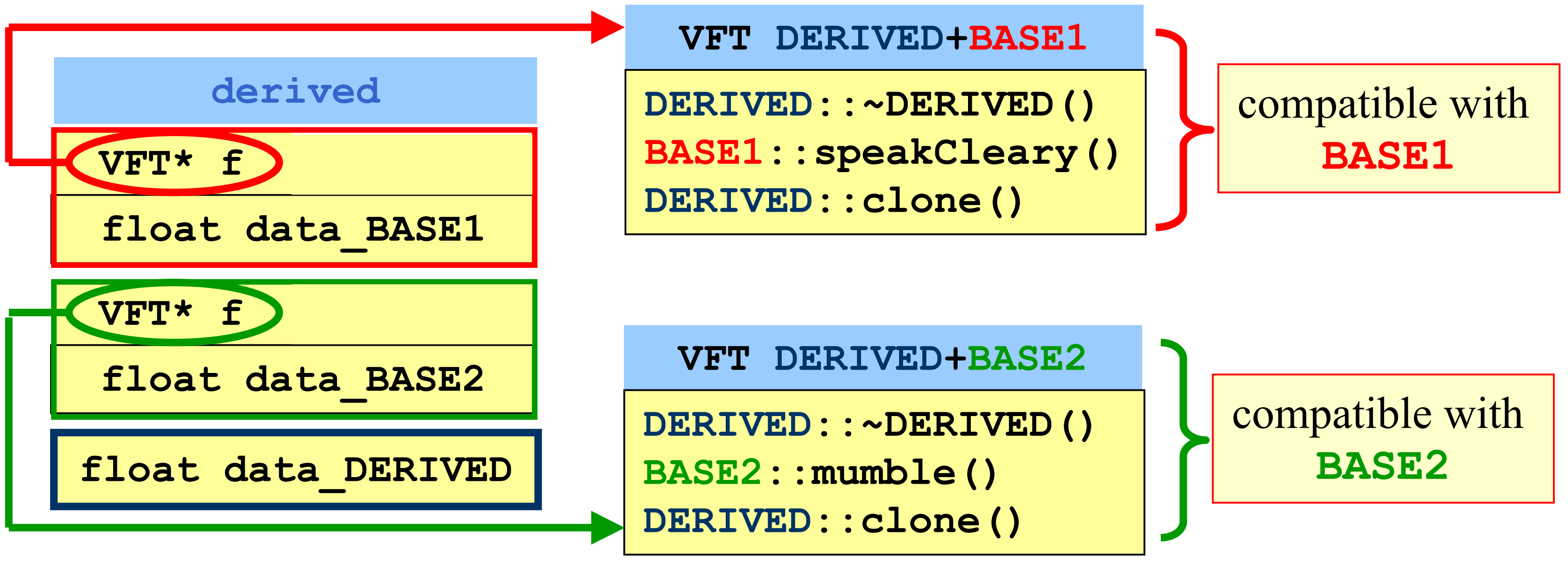
```
class DERIVED : public BASE1, public BASE2 {
public:
   DERIVED();
   virtual ~DERIVED();
   virtual DERIVED* clone() const;
protected:
   float data_DERIVED;
};
```

Note: DERIVED may either be casted to BASE1 or BASE2:

```
BASE1*  b1 = new DERIVED;
BASE2*  b2 = new DERIVED;
```

I.2.1. Subtype Polymorphism Method Dispatch - Multiple Inheritance II

- DERIVED needs access to BASE1's VFT **and** BASE2's VFT
- store pointer to *both* VFT's inside of object:



Observation:

1) A pointer to an instance of **`DERIVED`** is cast compatible with a pointer to an instance of **`BASE1`**.
2) If incremented by **`sizeof(BASE1)`**, a pointer to an instance of **`DERIVED`** is cast compatible with a pointer to an instance of **`BASE2`**.

I.2.1. Subtype Polymorphism Method Dispatch - Multiple Inheritance III

Adjustment of the **this** pointer during runtime is necessary:

```
BASE2* pbase2 = new DERIVED;
```

***pbase2** needs to behave like an instance of **BASE2** - increment **pbase2** by **sizeof(BASE1)**:

```
DERIVED tmp* = new DERIVED;
BASE2* pbase2 = tmp + sizeof(BASE1);
```

Nonpolymorphic access to members of **BASE2** still works (e.g. **pbase2->data_BASE2**)

BASE1::speakClearly is **inaccessible** now - unacceptable during virtual function calls !

Thus: if ***calling a virtual function*** the implicitly **this** parameter has to be adjusted in order to point to the beginning of the object - store suitable offset **o** inside object.

```
BASE2* theClone = pbase2->clone();
```

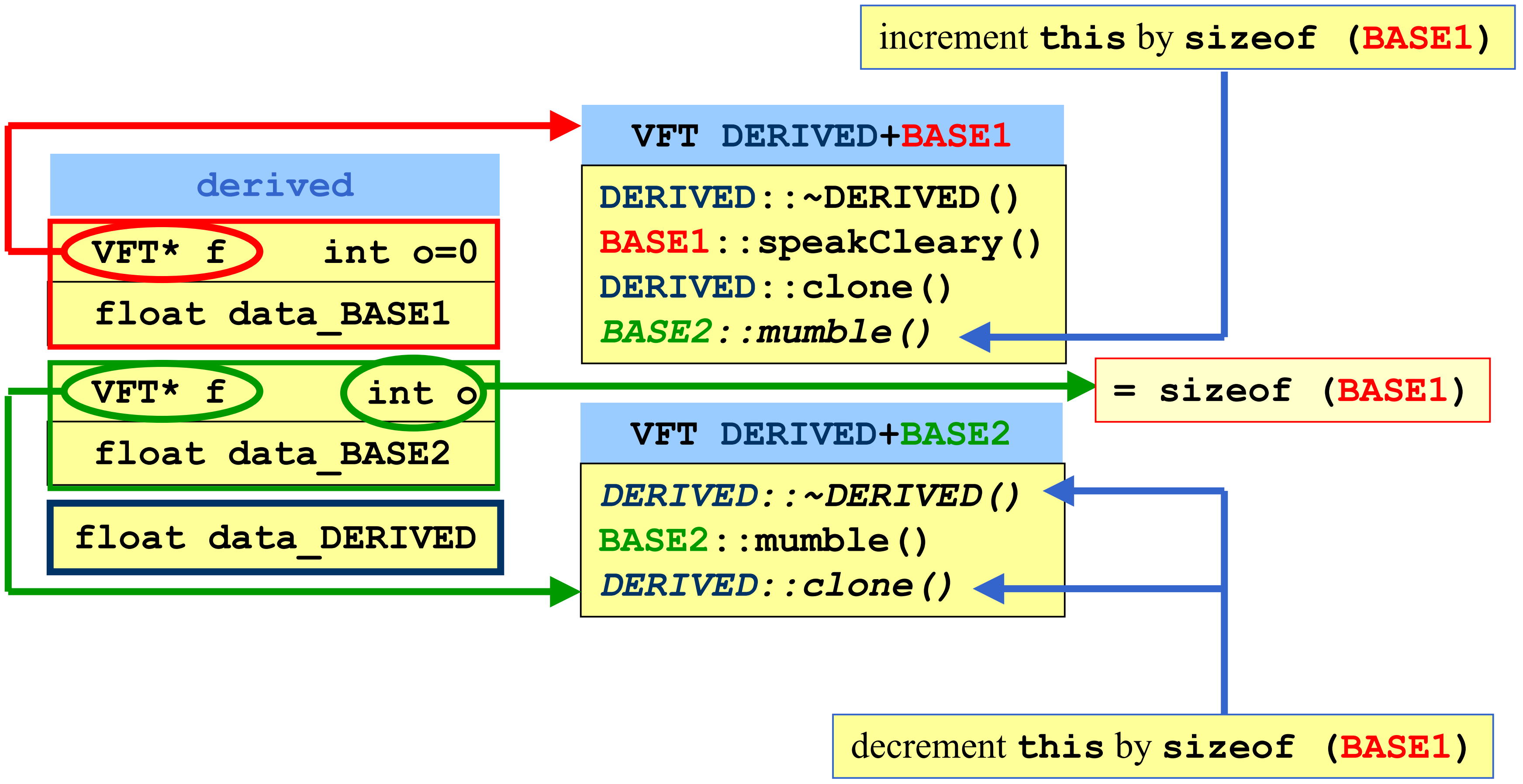
o is called **pointer casting offset**.

becomes

```
BASE2* theClone = pbase2->f->clone( pbase2 - pbase2->o );
```

call **DERIVED::clone -> this** has to be of type **DERIVED** !

I.2.1. Subtype Polymorphism Method Dispatch - Multiple Inheritance IV



This is how *cfront* implements virtual function calls.

Note 10: *Thunks*

!

The VFT technique ***penalizes all virtual function*** invocations regardless of whether the offset adjustment is necessary, both in the

- cost of one *extra access and addition* of offset and in
- the *increased size* of each virtual table slot.

- Many compilers (e.g. GCC) use **thunks** for virtual function calls.

A **thunk** is a small assembly stub that adjusts the **`this`** pointer with the appropriate offset and then jumps to the virtual function.

The thunk associated with the call to the **`DERIVED`** class destructor through a **`BASE2`** pointer look as follows:

```
Pbase2_dtor_thunk:
  this -= sizeof(BASE1);
  DERIVED::~DERIVED( this );
```

Address placed within the VFT either directly addresses the virtual function or points to the associated thunk.

I.2.1. Subtype Polymorphism Penalties

- standard implementation of virtual dispatch uses virtual function tables (**VFT**) which are basically tables of function pointers
- Initialization of pointer to **VFT** in constructor code
- in a typical implementation a virtual function call involves

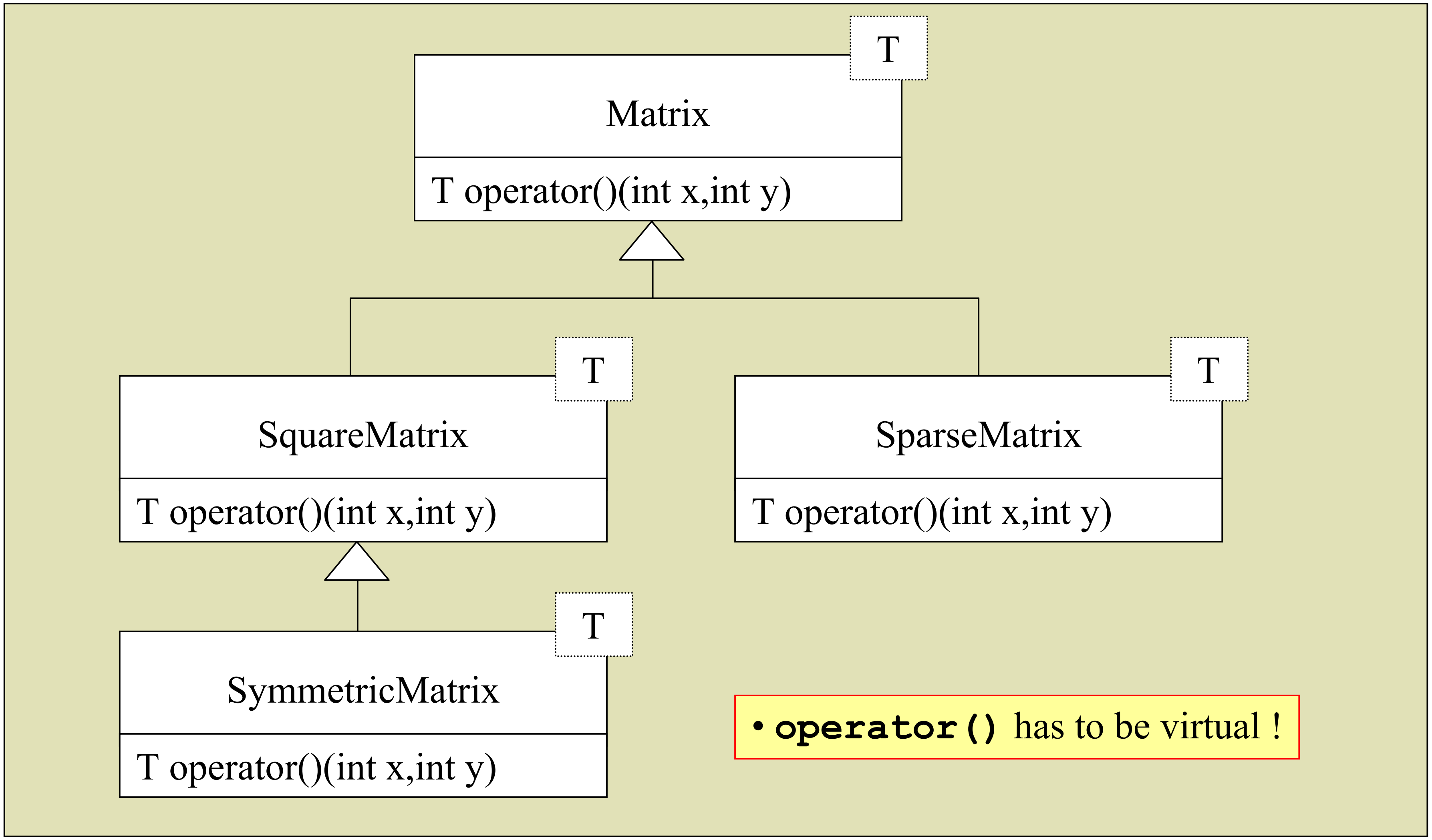
  1. two indirections to get the function pointer out of the VFT (get the VFT address and then the function address from the VFT)
  2. an extra indirection to get the pointer casting offset from the VFT (the offset is needed for multiple inheritance to work correctly)
  3. Offset addition (object plus the pointer casting offset)
  4. Function call

- small memory overhead (one VFT per class / at least one pointer to VFT + pointer casting offset per object.
- virtual function calls may prevent the compiler from applying various optimizations (the target of the call is unknown during compile time)

Avoid virtual functions, if they are ***small and used frequently*** !

Note 11: *Avoiding `virtual` Functions - Engines*

- Consider matrices with different storage schemes (dense,sparse,symmetric, etc.)
- Usual OO approach: build hierarchy (e.g. **`SquareMatrix`** is-a **`Matrix`**):


T
Matrix
T operator()(int x,int y)
T
SquareMatrix
T operator()(int x,int y)
T
SparseMatrix
T operator()(int x,int y)
T
SymmetricMatrix
T operator()(int x,int y)

- **`operator()`** has to be virtual !

Note 11: *Avoiding* `virtual` *Functions - Engines II*

```
template <typename T> class Symmetric {
 // Encapsulates storage info for symmetric matrices
};

template <typename T> class Square {
 // Encapsulates storage info for square matrices
};

template<class Engine_T> // Wrapper
class Matrix {
private:
  Engine_T engine;          // has an instance of an engine
};                          // (imports Engine_t's functionality)
```

```
// Example routine which takes any matrix structure
```

```
template<class Engine_T>
double sum(Matrix<Engine_T>& A);
```

```
// Example use ...
```

```
Matrix<Symmetric<double> > A;
sum(A);
```

Note 11: *Avoiding `virtual` Functions - Engines III*

- Delegate functionality to the **`engine`**

```
template <typename T> class Symmetric {
   typedef T Element_T;
   inline Element_t& operator()(int x,int y) { ... }
   bool isPositiveDefinite() { ... }
};

template<class Engine_T>
class Matrix {
   typedef typename Engine_T::Element_T Element_T;
   inline Element_T& operator()(int x,int y) {
     return engine(x,y);
   }
   bool isPositiveDefinite() {
      return engine.isPosisitiveDefinite();
   }
private:
   Engine_T engine; // instance of engine
};
```

- Matrix *subtypes* types must have the same member functions
- Some members only make sense for a subset of all subtypes (e.g. `isPositiveDefinite`)

Note 11: *Avoiding* `virtual` *Functions - Engines IV*

- What about **`Square`** ?
- Needs to have **`isPositiveDefinite`** as well:

```
template <typename T> class Square {
  typedef T Element_T;
  inline Element_t& operator()(int x,int y) { ... }
  bool isPositiveDefinite() {
    throw makeError("Method not defined for square matrices !");
  }
};
```

Matrix base class needs to have the union of all methods provided by the subtype

if you have to deal with a huge class hierarchy, adding a single method to a subtype means to change every subtype as well !

Note 12: *Avoiding `virtual` Functions - Barton/Nackman Trick I*

```
// Base class takes a template parameter. This parameter
// is the type of the class which derives from it.
template<class Leaftype_T>
class Matrix {
public:
  Leaftype_T& asLeaf() { return static_cast<Leaftype_T&>(*this);}
  // delegate to leaf
  double operator()(int i, int j) { return asLeaf()(i,j); }
};

class SymmetricMatrix : public Matrix<SymmetricMatrix> { ... };
class UpperTriMatrix  : public Matrix<UpperTriMatrix>  { ... };
```

```
// Example routine which takes any matrix structure
```

```
template<class Leaftype_T>
double sum(Matrix<Leaftype_T>& A);
```

```
// Example use ...
```

```
SymmetricMatrix A;
sum(A);
```

Note 12: *Avoiding* `virtual` *Functions - Barton/Nackman Trick II*

- More convenient inheritance-hierarchy approach
- Base class still delegates functionality to the leaf classes
- Members can selectively specified / specialised in the leaf class

```
template<class Leaftype_T>
class Matrix {
public:
  Leaftype_T& asLeaf() { return static_cast<Leaftype_T&>(*this);}
  // delegate to leaf
  double operator()(int i, int j) { return asLeaf()(i,j); }
  bool isPositiveDefinite() {
    throw makeError(“Method not defined for square matrices !”);
  }
};

class SymmetricMatrix : public Matrix<SymmetricMatrix> {
  bool isPositiveDefinite() { ... }
};
```

Overview

- Classes
- *Polymorphism*
  - Inheritance
  - *Overloading*
  - Conversions
  - Templates
- Further Topics
- References

I.2.2. Overloading

Overloading

In C++ functions are identified by their name *and* their **argument types**. Having multiple functions of the same name but different argument types is called **overloading**.

```
void swap(int& a,int& b) {
   int t;
   t=a; a=b ; b=t;
}

void swap(double& a,double& b) {
   double t;
   t=a; a=b ; b=t
}
```

- Overloading on the return type is not supported, because a function may be called just for its side effects:

e.g. `foo();` instead of `ResultType t = foo();`

I.2.2. Overloading

- Besides function overloading C++ supports **operator overloading**.

Binary Operator Overloading

A binary operator is an infix function that may be written between their arguments rather before them, as in the case for ordinary functions.

I.2.2. Overloading - A Sample Class

• Sample class to represent complex numbers:

```
class Complex {
private:
  double re;
  double im;
public:
  Complex(double r=0.0,double i=0.0) : re( r ),   im (i)      {}
  Complex(const Complex& rhs)         : re(rhs.re),im(rhs.im) {}

  const double& real() const { return re; }
  const double& imag() const { return im; }

  double& real() { return re; }
  double& imag() { return im; }
};
```

I.2.2. Overloading - Assignment Operators

- The assignment operator (**`operator=`**) can be overloaded. *It has to be implemented as a class member function*:

```
Complex& Complex::operator=(const Complex& rhs) {
  re = rhs.real() ; im = rhs.imag();
  return *this;
}
```

- To allow **daisy chaining** (e.g. **`a = b = c`**) , assignment operators return references.

If not defined, the compiler *automatically creates* an **assignment operator** that performs a memberwise copy.

- Do not forget to distinguish between copy constructor and assignment operator

`Complex c2 = c1;` $\longrightarrow$ `Complex c2(c1);`

`Complex c2; c2 = c1;` $\longrightarrow$ `c2.operator=(c1);`

I.2.2. Overloading - Arithmetic Operator Overloading

- C++ allows arithmetic operators (`+=`, `-=`, `*=`, `/=`, `%=` and their binary cousins `+`, `-`, `*`, `/`, `%`) to be overloaded

- Overloaded operators are syntactic sugar allowing to write code like

```
Complex a,b,c,d;
a = b + c * d;
```

- **Operator precedence** cannot be changed, thus, `b + c * d` always is identical to `b + (c * d)`

- Inventing **new operators** is impossible (e.g. `(c ** d)`)

I.2.2. Overloading - Arithmetic Assignment Operators

- **Arithmetic assignment operators** (`+=`, `-=`, `*=`, `/=`, `%=`) are overloaded through class member functions

```
const Complex& Complex::operator+=(const Complex& rhs) {
  re += rhs.real() ; im += rhs.imag();
  return *this;
}
```

- return **`const`** object to disallow expressions like **`(a += b)++`**

```
Complex a,b;
a+=b;
```

transformed by compiler into →

```
Complex a,b;
a.operator+=(b);
```

I.2.2. Overloading - Binary Arithmetic Operators & Member Functions

Binary operators shouldn't be class members - especially if the operation should be **commutative**.

```
Complex Complex::operator+(const Complex& a) const {
  return Complex(*this) += a ;
}
```

- Due to the presence of a constructor from **`double`**, a value of type **`double`** ***implicitly*** can be ***converted*** into a value of type **`Complex`**. Thus, this operator implicitly defines how to add a **`double`** to a **`Complex`** - but ***not*** how to add a **`Complex`** to a **`double`**:

```
Complex a,b;
a = b + 1.0
```

transformed by compiler into

```
Complex a,b;
a = b.operator+(Complex(1.0))
```

Implicit conversion from **`double`** to **`Complex`**

```
Complex a,b;
a = 1.0 + b
```

transformed by compiler into

```
Complex a,b;
a = (1.0).operator+(b)
```

Nonsense !

I.2.2. Overloading - Binary Arithmetic Operators & Free Functions

• **Binary operators** should not be class members, but free functions:

```
Complex operator+(const Complex& a,const Complex& b) {
  Complex tmp;
  tmp.re = a.real() + b.real();
  tmp.im = a.imag() + b.imag();
  return tmp;
}
```

```
Complex a,b;
a = 1.0 + b
```

transformed by compiler into

```
Complex a,b;
a = operator+(Complex(1.0),b)
```

To gain access to a class' private data members, a binary operator can be made a **`friend`** function, or it can be based on the implementation of **`operator+=`** :

```
Complex operator+(const Complex& a,const Complex& b) {
  return Complex(a)+=b;
}
```

Binary operators have to **return their result *by value***, because a new object is created !

I.2.2. Overloading - Penalties

At first glance the following code seems harmless:

```
Complex a,b,c;
a = b + c;
```

$\longrightarrow$

```
Complex a,b,c,d;
a.operator=(operator+(b,c));
```

Unfortunately we have to deal with ***two*** **temporary objects**:

- `operator+` creates a temporary (local object)
- another temporary is created to return the result *by value*:

```
Complex tmp;
```

```
return tmp;
```

Constructor needs to be called *two* times - **very expensive** for e.g. matrices

- implicit conversion may introduce additional temporaries (*see next section*), e.g.

```
Complex a,b;
a = b + 1;
```

- **Overload resolution** consumes compile time

Note 13: *Named Return Value (NRV) Optimization*

- Some compilers perform the **Named Return Value Optimization**.

```
Complex operator+(const Complex& a,const Complex& b) {
  Complex tmp;
  tmp.re = a.real() + b.real();
  tmp.im = a.imag() + b.imag();
  return tmp;
}

Complex c = a + b;
```

- **two** temporaries get created: local temporary **`tmp`** and temporary to carry result

NRV

- eliminates local temporary, copy constructor call, and destructor call
- does not apply if multiple return statements return objects of different name
- probably problematic (copy constructor has side effects)

Note 13: *Named Return Value (NRV) Optimization II*

```
Complex operator+(const Complex& a,const Complex& b) {
  Complex tmp;
  tmp.re = a.real() + b.real();
  tmp.im = a.imag() + b.imag();
  return tmp;
}
```

```
void operator+(Complex& __result, const Complex& a,const Complex& b) {
  Complex tmp;                         // this is just raw memory !
  Complex::Complex(tmp);               // call default constructor
  //.. compute - store result in tmp
  Complex::Complex(__result, tmp ); // copy constructor
  Complex::~Complex(tmp);
  return;
}
```

↓ NRV (eliminates *local* temporary `tmp`)

```
void operator+(Complex& __result, const Complex& a,const Complex& b) {
  Complex::Complex(__result);        // call default constructor
  // .. compute - store result in __result
  return;
}
```

Notice, that **`operator+`** expects **`__result`** to be raw memory; not an initialized object !

Note 14: *Optimizing away Temporary Return Values*

```
Complex c = a + b;
```

```
Complex c;          // Raw memory - no constructor call here !
Complex retval;                      // Temporary to hold return value
operator+(retval,a,b);               // compute result
Complex::Complex(c, retval );        // copy construct c from retval
Complex::~Complex(retval)            // destruct temporary
```

optimized

```
Complex c;
operator+(c,a,b);
```

No temporary needed !

```
c = a + b;
```

```
Complex retval;                      // Temporary to hold return value
operator+(retval,a,b);               // compute result
                                     // expects retval to be raw memory
operator=(c, retval );               // copy assign c from retval
Complex::~Complex(retval);           // destruct temporary
```

```
Complex::~Complex(c);// must call destructor here
operator+(c,a,b);    // semantically Complex::Complex(c, a + b );
```

assignment vs. **destruction + copy construction** - cannot remove temporary

Note 15: *Computational Constructors*

? **What, if your compiler does not perform the NRV ?**

- **Solution 1**: do computation within constructor:

```
class Complex {
private:
  double re;
  double im;
public:
  Complex(double r=0.0,double i=0.0) : re( r ),   im (i)      {}
  Complex(const Complex& rhs)        : re(rhs.re),im(rhs.im) {}
  Complex(const Complex& a,const Complex& b) : re(a.re + b.re),
                                               im(a.im + b.im) {}
  ...
};
```

```
Complex operator+(const Complex& a, const Complex& b) {
  return Complex(a,b);                // local temporary here, but most
}                                     // compilers optimize it away
```

- How to perform subtraction now ? This scheme leads to ambiguous constructor definitions !
- Introduce a **Tag Class**

Note 15: *Computational Constructors*

```
struct Add {};
struct Sub {};

class Complex {
  //...
public:
  Complex(double r=0.0,double i=0.0) : re( r ),   im (i)      {}
  Complex(const Complex& rhs)        : re(rhs.re),im(rhs.im) {}
  Complex(const Complex& a,const Complex& b,const Add&) :
      re(a.re + b.re),
      im(a.im + b.im) {}
  Complex(const Complex& a,const Complex& b,const Sub&) :
      re(a.re - b.re),
      im(a.im - b.im) {}
  // ...
};
```

```
Complex operator+(const Complex& a, const Complex& b) {
  return Complex(a,b,Add());
}
Complex operator-(const Complex& a, const Complex& b) {
  return Complex(a,b,Sub());
}
```

Note 16: *Reuse Arithmetic Assignment Operqators*

! Consider reusing arithmetic assignment operators. Does this make sense ?

```
const Complex& Complex::operator+=(const Complex& rhs) {
  re += rhs.real() ; im += rhs.imag();
  return *this;
}
```

No temporary here !

```
Complex operator+(const Complex& a,const Complex& b) {
  return Complex(a)+=b;
}
```

Anonymous temporaries get optimized away by most compilers - **but you need to provide a copy constructor !**

But: we still need a temporary to carry the return value !

Note 17: *op++ vs. ++op*

! Prefer **++op** / **--op** if possible !

Semantics of **op++** / **op--**: Return the **current value** of **op** - then increment / decrement it

```
X X::operator++(int) {
  X current(*this);
  //... Computation to increment *this (e.g. perform ++(*this) )
  return current;
}
```

Local temporary needed (independent on **X**)

Semantics of **++op** / **--op**: Return the **incremented / decremented value** of **op**

```
X X::operator++() {
  //... Computation to increment *this
  return *this;
}
```

No local temporary needed !!

Note 18: *Arithmetic Assignment Operators to Replace Binary Op's*

Observation:
Usually arithmetic assignment operators can be implemented without introducing a local temporary.

```
Complex a,b;
result = a + b;  // local temporary gets created !
```

$$\Big\downarrow \text{ transformed by hand}$$

```
Complex a,b;
result  = a;  // operator=  - no local temporary
result += b;  // operator+= - no local temporary
```

... unfortunately, we loose a lot of beauty here ...

Overview

- Classes
- *Polymorphism*
  - Inheritance
  - Overloading
  - *Conversions*
  - Templates
- Further Topics
- References

I.2.3. Conversions

- `C++` supports user defined / automatic conversion
- Unary constructors can be used to perform **implicit conversion** from the type of ist first parameter to the type of its class

```
Complex a,b;
a = b + 1;
```

Is converted into **`Complex`** through unary constructor

- implicit conversion *introduces temporary* objects
- use keyword `explicit` to disallow implicit conversion, e.g.:

```
explicit Complex(double r=0.0,double i=0.0) : re(r), im(i) {}
```

- unary constructors **cannot** be used to
  1.) convert a user type into a fundamental type (e.g. `int`, `double`)
  2.) convert a new class to an existing one (need to change existing class)

I.2.3. Conversions - Conversion Functions

- conversion functions are special class member functions, e.g.:

```
Complex::operator double() const {
   return real();
}
```

return type specified by operator name only

- **Caution:** ambiguities are possible

```
Complex a = b + 5
```

`operator+(Complex,Complex)` or built-in **`operator+(double,double)`** ?

- an implicit conversion sequence is one of the following forms

  1.) a standard conversion sequence
  2.) a user-defined conversion sequence
  3.) an ellipsis conversion sequence

I.2.3. Conversions - Standard Conversions I

1. Lvalue-to-rvalue conversion

Exact Match

An lvalue is an expression that may be used to the left of an assignment operator. An rvalue is an expression that may be used to the right of an assignment operator: it represents a value that does not have an address and that cannot be modified.

```
int& f() { ... }
int x = f();
```

2. Array to pointer conversion

Exact Match

An array of `T`'s can be converted into a pointer to `T`

```
int i[20];
int* j=i;
```

3. Function to pointer conversion

Exact Match

An lvalue of function type `T` can be converted to an rvalue of type “pointer to `T`.” The result is a pointer to the function.

```
int& (g*)() = f;
```

I.2.3. Conversions - Standard Conversions II

4. Qualification conversion

Exact Match

A qualification conversion adds **`const`** or **`volatile`** qualifications to pointers.

```
int f(const int& a) { return a + 3; }
int i;
f(i);
```

5. Floating point promotion

Promotion

An rvalue of type **`float`** can be converted to an rvalue of type **`double`**.

```
float  f;
double d;
d = f;
```

6. Floating point conversion

Conversion

An rvalue of floating point type can be converted to an rvalue of another floating point type.

I.2.3. Conversions - Standard Conversions III

7. Integral promotion

Promotion

- An rvalue of type **`char`**, **`signed char`**, **`unsigned char`**, **`short int`**, or **`unsigned short int`** can be converted to an rvalue of type **`int`** if int can represent all the values of the source type; otherwise, the source rvalue can be converted to an rvalue of type **`unsigned int`**.
- An rvalue of type **`wchar_t`** or an enumeration type can be converted to an rvalue of the first of the following types that can represent all the values of its underlying type: **`int`**, **`unsigned int`**, **`long`**, or **`unsigned long`**.
- An rvalue for an integral bit-field can be converted to an rvalue of type **`int`** if **`int`** can represent all the values of the bit-field; otherwise, it can be converted to **`unsigned int`** if **`unsigned int`** can represent all the values of the bit-field. If the bit-field is larger yet, no integral promotion applies to it. If the bit-field has an enumerated type, it is treated as any other value of that type for promotion purposes.

I.2.3. Conversions - Standard Conversions IV

8. Integral conversion

Conversion

An rvalue of an integer type can be converted to an rvalue of another integer type. An rvalue of an enumeration type can be converted to an rvalue of an integer type.

9. Floating-integral conversion

Conversion

- An rvalue of a floating point type can be converted to an rvalue of an integer type. The conversion truncates; that is, the fractional part is discarded.
- An rvalue of an integer type or of an enumeration type can be converted to an rvalue of a floating point type. The result is exact if possible.

10. Pointer conversion

Conversion

- An rvalue of type pointer to *cv* **`T`**, where **`T`** is an object type, can be converted to an rvalue of type pointer to *cv* **`void`**.
- An rvalue of type pointer to *cv* D, where D is a class type, can be converted to an rvalue of type pointer to *cv* B, where B is a base class of D.

cv=[**`const`**,**`volatile`**] - (may be empty)

I.2.3. Conversions - Standard Conversions V

11. Pointer to member conversions

Conversion

An rvalue of type pointer to member of B of type cv T, where B is a class type, can be converted to an rvalue of type pointer to member of D of type cv T, where D is a derived class of B.

12. Boolean conversions

Conversion

An rvalue of arithmetic, enumeration, pointer, or pointer to member type can be converted to an rvalue of type bool. A zero value, null pointer value, or null member pointer value is converted to false; any other value is converted to true.

I.2.3. Conversions - Conversion Sequences

Standard Conversion Sequence

A **standard conversion sequence** is either the Identity conversion by itself (that is, no conversion) or consists of one to three conversions from the categories Lvalue Transformation, Qualification Adjustment, Promotion, and Conversion. At most one conversion from each category is allowed in a single standard conversion sequence.

User-defined Conversion Sequence

A **user-defined conversion sequence** consists of an initial standard conversion sequence followed by a user-defined conversion followed by a second standard conversion sequence.

Only ***one*** user-defined conversion per conversion sequence !

Ellipsis Conversion Sequence

An **ellipsis conversion sequence** occurs when an argument in a function call is matched with the ellipsis parameter specification of the function called.

Note 19: *Overloading on Return Type*

Suppose we want to overload functions based on the return type:

```
class A { ... };
class B { ... };

A foo(int x) { ... }  // 1
B foo(int x) { ... }  // 2

A a = foo(3); // calls 1
B b = foo(3); // calls 2
```

We can't do that in C++. **But we can fake it !**

Idea: provide a single function **`foo`** that returns a value that can be converted into a value of type **`A`** or a value of type **`B`** respectively - let **`foo`** return a **proxy object** !

Note 19: *Overloading on Return Type*

```
A fooA(int x) { ... }
B fooB(int x) { ... }

class C {
  int x_m;
public:
  C(int x) : x_m(x) {}
  operator A() const { return fooA(x_m); }
  operator B() const { return fooB(x_m); }
};

C foo(int x) { return C(x); }

A a = foo(3); // calls C::operator A() - calls fooA
B b = foo(3); // calls C::operator B() - calls fooB
```

Calling a function just for its side effects is not possible anymore !

Overview

- Classes
- *Polymorphism*
  - Inheritance
  - Overloading
  - Conversions
  - *Templates*
- Further Topics
- References

I.2.4. Templates

- Allows a part of a program to be used with different types
- Types as additional parameters to function / class / or member function declarations
- Substitution of type parameters during compile time (*template instantiation*)
- Ideal for container classes (e.g. Stack, List, Tree - *STL*)
- C++ supports
    - function templates
    - class templates
    - member function templates

I.2.4. Templates - Function Templates

```
template <typename T>
void copyArray(const T& source,T& dest,int size) {
  for (int i=0 ; i<size ; i++)
    dest[i]=source[i];
}
```

• `T` is a **type parameter**

• no code is generated until template gets **instantiated**

Instantiation

Instantiation is the process of binding actual types and expressions to the associated type parameters of the template.

```
int aI[100], bI[100];
...
copyArray(aI,bI,100);
```

• Using the **`copyArray`** template the instantiation process binds `T` to `int*` and creates a program text instance of **`copyArray`** (by possibly overloading existing versions).

I.2.4. Templates - Multiple Type Parameters

```
template <typename T>
void copyArray(const T& source,T& dest,int size) {
  for (int i=0 ; i<size ; i++)
    dest[i]=source[i];
}
int aI[100], bI[100];
double aD[100],bD[100];
...
copyArray(aI,bI,100); // o.k. instantiation with T=int*
copyArray(aD,bD,100); // o.k. instantiation with T=double*
copyArray(aI,bD,100); //error: source and dest must be of same type
```

• Introduce second type parameter to make this work:

```
template <typename T1,typename T2>
void copyArray(const T1& source,T2& dest,int size) {
  for (int i=0 ; i<size ; i++)
    dest[i]=source[i];
}
```

Constraint: elementwise conversion from **T2** to **T1** must exist !
But: there are more constraints here !

I.2.4. Templates - Explicit Qualification

```
template <typename T>
T average(const T& a,const T& b) {
  return (a + b) / 2;
}
int a,b;
double c,d;
...
average(a,b); // o.k. instantiation with T=int
average(c,d); // o.k. instantiation with T=double
average(a,c); // error: source and dest must be of equal Type T
```

- **problem:** type cannot be deduced from arguments (`int` or `double` ?)
- use **explicit qualification**

```
average<int>(a,c);    // works ! Floating point-integral
average<double>(a,c); // conversion applies here !
```

- explicit qualification can be **mandatory**

```
template <typename R,typename A>
R convert_A2R(A a) { return R(a); }
```

I.2.4. Templates - Class Templates

• Classes may be templates:

```
template <typename T>
class dArray {
private:
  int size;
  T* data;
public:
  dArray(int s) : size(s) { a = new T[size]; }
  ~dArray()               { delete [] a; }
  T& operator[](int s)             { return data[s]; }
  const T& operator[](int s) const { return data[s]; }
};
```

• Like function templates, only instantiated if needed

```
dArray<int> aI(100),bI(100);
dArray<double> aD(100),bD(100)
```

• Does **`copyArray(aI,bI)`** work ?

Yes: **`dArray<int>`** provides a *subscript operator* and copy from **`int`** to **`int`** is trivial !

I.2.4. Templates - Non-type Parameters I

- What if we don't want to use dynamic memory allocation because the size of the array is always known during compile time ?

- Make **`size`** a template parameter:

```
template <typename T,int size>
class sArray {
private:
  T data[size];
public:
  T& operator[](int s)                  { return data[s]; }
  const T& operator[](int s) const { return data[s]; }
};
```

- Still works with **`copyArray`** but we can utilize that **`size`** is known at compile time:

```
template <typename T1,typename T2,int s>
void copyArray(sArray<T1,s>& source,sArray<T2,s>& dest,int S=s){
  for (int i=0 ; i<S ; i++)
    dest[i]=source[i];
}
```

I.2.4. Templates - Non-type Parameters II

- A non-type *template-parameter* shall have one of the following (optionally *cv-qualified*) types:
  - integral or enumeration type,
  - pointer to object or pointer to function,
  - reference to object or reference to function,
  - pointer to member.

- non-type parameters **shall not** be declared to have
  - floating point type (e.g. **`float`**, **`double`**)
  - class type
  - void type

```
template <double& a> struct A {...};  // is o.k. !
template <float* b>  struct B {...};  // o.k. as well !
Template <double d>  struct C {...};  // error !
```

I.2.4. Templates - Template Template Parameters

- templates as template parameter

```
template <class T, template <class U> class C>
class Group {
  C<T> container;
};
```

- a template template argument shall be the name of a *class templates* !
- function templates are not allowed as template template arguments

- instantiation:

```
Group< double, std::list >  a; // use class template std::list.
Group< int, std::vector >   b; // use class template std::vector
```

I.2.4. Templates - Member Templates I

- a class member function can be a template

```
template <typename A>
class foo {
private:
  A value;
public:
  foo(const A& v) : value(v) {}
  template <typename T>
  void setValue(const T& v) { value = v; }
};
```

- definition of class member template outside class definition

```
template <typename A>
template <typename T>
void foo<A>::setValue(const T& v) { value = v; }
```

- a class template may contain member templates and/or other class templates
- class member templates may not be virtual
- local classes shall not contain member templates

I.2.4. Templates - Member Templates II

- a specialization of a member template does not override a virtual function from a base class:

```
class BASE {
  virtual void foo(int);
};

class DERIVED : public BASE {
  template <typename T>
  void foo(T);                   // does not override BASE::foo(int)
  void foo(int i){ foo<>(i); } // overriding function that calls
                               // the template instantiation
};
```

- calling template conversion functions:

```
struct A {
  template <typename T> operator T*() {return 0;}
};
template <> A::operator char*() { return 0; } // specialization
template A::operator void*();             // explicit instantiation

A a;
int *i = a.operator int*();  // instantiation of member template
```

I.2.4. Templates - Default Template Arguments

- a default template argument may be specified for any type of template parameter

```
template <typename T,int size = 1000>
class sArray { ... };
```

- may only be specified in a class template declaration / definition
- if a template parameter has a default argument, all subsequent template parameters shall have a default argument as well.

```
template <typename T = double, int size>  // error !
class sArray {... };
```

- The scope of a *template-parameter* extends from its point of declaration until the end of its template.

```
template<class T, T* p, class U = T> class X { ... };
```

- caution: the first non-nested **>** is taken as the end of template parameter list

```
template <int s = 2 > 42 > ...    // error !
template <int s = (2 > 42) > ... // o.k.
```

I.2.4. Templates - Explicit Instantiation

- usually template instantiation is done implicitly by the compiler
- it also can be done explicitly by the programmer

```
template class sArray<int>;                         // class
template class int& sArray<int>::operator[](int);   // member
template double convert_A2R<double,int>(int);       // function
```

- if a class gets explicitly instantiated, *every member gets instantiated as well*
- explicit instantiation can save compile time and link time

I.2.4. Templates - Specialization

- class templates, function templates and member templates may be explicitly specialized

```
template <typename T>
struct A {
   void bar(const T&);
   static void foo() { cout << "Hello !" << endl; }
};

template <>
struct A<char*> {
   static void foo() { cout << "World" << endl; }
};

A<int>::foo();    // prints "Hello !"
A<char*>::foo(); // prints "World"
```

- nothing gets implicitly instantiated during explicit specialization

```
A<char*>::bar("Hello"); // error ! Method bar not defined in
                        // specialization of A
```

- implicitly generated and explicitly specialized classed do not need to be related !

I.2.4. Templates - Specialization II - Specialization of Class Members / Member Templates

- Specialization of members is allowed even if they have been defined in class definition

```
template <typename A>
class foo {
  template <typename T>
  T void id(const T& t) {
    return t;
  }
  void f(const A& a);
};

template <>                              // specialization of member
void foo<int>::f(const int& a);

template <>                              // specialization of member template
template <>
int void foo<int>::id<double>(const double& d) {
  return d*2;
};
```

I.2.4. Templates - Specialization III - Point of Declaration / Point of Instantiation

• placement of explicit specialization declaration is important !

```
template <typename T>
void sort(sArray<T>& a) { ... }; // point of declaration

void foo(sArray<string>& a) {
  sort(a); // point of instantiation
           // implicit instantiation of sort<sting>, from here
           // on sort<string> is defined !
};

template<> void sort<string>(sArray<string>& a); // error !
```

I.2.4. Templates - Specialization IV - Function Templates - Inline / Static Vars

• an explicit specialization of a function template is inline only if it is explicitly declared to be

```
template <typename T> inline T twice(const T& t) {
  return 2 * t;
}

template <> int twice<int>(const int& t) { // not inline !
  return t << 1;
}
```

• each function template specialization has its own copy of any static variables

```
template <typename T> void test(T a) {
  static int j;
  cout << j << " ";
  j=a;  cout << j << endl;
}

double d=3.33; int i=2;
test(d);  // prints "0 3"
test(i);  // prints "0 2"
test(d);  // prints "3 3"
```

I.2.4. Templates - Class Template Partial Specialization

- **Primary Template (unspecialized)**

```
template <typename A,typename B,int i> class foo {};
```

- **Partial Specialization - B=A* - B is a pointer to A**

```
template <typename Q,int j> class foo<Q,Q*,j> {};
```

- **Partial Specialization - A is a pointer**

```
template <typename Q,typename R,int j> class foo<Q*,R,j> {};
```

- **Partial Specialization - A is an `int` , B is a pointer**

```
template <typename Q,int j> class foo<int,Q*,j> {};
```

- **Full Specialization `A=int B=int i=3`**

```
template <> class foo<int,int,3> {};
```

```
foo<int,int*,5> a; // selects third specialization
```

I.2.4. Templates - Partial Specialization - Member Templates

```
template <typename A> class foo {
  template <typename B> class bar {
    template <typename C> class zap { ... };
  };
};
```

- full specialization - all template parameter get bound

```
template <> template<> template<>
class foo<int>::bar<int>::zap<int> {
};
```

- partial specialization - few template parameter get bound (outer to inner)

```
template <> template<typename B> template<typename C>
class foo<int>::bar<B>::zap<C> {
};
```

- specializing a class template / class member template without specializing enclosing templates is illegal

```
template <typename A> template<typename B> template<>
class foo<A>::bar<B>::zap<int> {    // ILLFORMED !
};
```

Note 20: *How to specialize inner-class template I*

```
template <typename A> template<typename B> template<>
class foo<A>::bar<B>::zap<int> { ...};  // ILLFORMED
```

```
template <typename A,typename B,typename C>
struct _zap { ... };

template <typename A,typename B>
struct _zap<A,B,int> { ... }; // Allowed specialization !

template <typename A>
struct foo {
   typedef int Value_t;
   template<typename B>
   struct bar {
     template<typename C>
     struct zap {
       typedef _zap<A,B,C> Ret;
     };
   };
};

foo<int>::bar<double>::zap<int>::Ret a; // Selects partial
                                        // specialization of _zap
```

Problem: `_zap` is not in *scope* of `foo` and `bar`

But: `_zap` can access all static information that is exported by `foo` and `bar` !

Note 20: *How to specialize inner-class template II*

```
template <typename A> foo;   // Declare foo such that we can use it
                             // within _zap
template <typename A,typename B,typename C>
struct _zap {
  typedef typename foo<A>::Value_t Value_t;
};

template <typename A,typename B>
struct _zap<A,B,int> {
  typedef typename foo<A>::Value_t Value_t;
};

template <typename A>
struct foo {
   typedef int Value_t;       // 'Export' a type definition
   template<typename B>
   struct bar {
     template<typename C>
     struct zap {
       typedef _zap<A,B,C> Ret;
     };
   };
};
```

Import type definition from `foo`

Overview

- Classes
- Polymorphism
  - Inheritance
  - Overloading
  - Conversions
  - Templates
- *Further Topics*
- References

I.3.1. Aliasing and the `restrict` keyword

```
void foo(double y[], const double* a, int n ) {
  for( int i=0; i<n; i++ )
    y[i] = 1.0 - *a;
}
```

Assuming the **`*a`** remains invariant during execution of the loop, we can perform the following optimization:

```
void foo(double y[], const double* a, int n ) {
  float temp = 1.0 - *a;
  for( int i=0; i<n; i++ )
    y[i] = temp;
}
```

Problem: If **`y`** and **`a`** overlap, this optimization changes the semantics !

I.3.1. Aliasing and the `restrict` keyword

```
void foo(double y[], const double* a, int n ) {
  for( int i=0; i<n; i++ )
    y[i] = 1.0 - *a;
}
```

```
double ar[100];
ar[50] = 1.0;
foo( ar, &ar[50], 100)
```

- first 50 iterations set **`ar[0]`**...**`ar[49]`** to **`0.0`** (**`1 - ar[50]`**)
- the 51st iteration changes **`ar[50]`** from **`1.0`** to **`0.0`**.
- Since **`*a`** is aliased to **`ar[50]`** subsequent iterations set **`ar[51]`**...**`ar[99]`** to **`1.0`**

The optimizer needs to be conservative !
Optimization is not performed !

I.3.1. Aliasing and the `restrict` keyword

- **Solution:** sign a contract with the compiler !

```
void foo(double* restrict y, const double* a, int n ) {
  for( int i=0; i<n; i++ )
    y[i] = 1.0 - *a;
}
```

- **`restrict`** is a type specifier like **`const`**, **`volatile`**
- it qualifies the pointer **`y`** - not the target
- **`restrict double*`** is illegal !

For the lifetime of pointer **`y`**, only **`y`** or pointers copied directly or indirectly from **`y`** will be used to reference the sub-object pointed to by **`y`**.

I.3.1. Aliasing and the `restrict` keyword

- `restrict` must not be part of the interface

```
void foo(double* y, const double* a, int n ) {
  double* restrict local_y = y;
  for( int i=0; i<n; i++ )
    local_y[i] = 1.0 - *a;
}
```

- `restrict` **is not part of the C++ standard** (however, many compilers support it)
- you therefore may prefer using it in the body of a function only and keep the interface clean
- however, making it visible in the interface tells the client not to pass aliased pointers
- **example**: `memcpy` aliasing not allowed by standard !

Notice, that there may exists hidden invariants (e.g. hidden by an inline function or a template)

I.3.2. Advanced Inlining - Virtual functions

• Often asked: “**Does inlining of virtual functions make sense ?**”

```
struct foo {
   inline virtual void test() { ... }
};

struct bar : public foo {
   inline virtual void test() {
      foo::test(); // inlining
   }
};

inline void no_inline(const foo* f) {
   f->test();        // no inlining
}

...
bar b;
b.test();           // inlining
no_inline(&b);
```

Yes, because sometimes the target of the call is resolvable during compile-time !

I.3.2. Advanced Inlining - Cross-Call Optimization

- Inlining may enable additional optimizations

```
double foo = 90.0;
//.. code that doesn't change foo
float bar = sin(foo);
```

```
double foo = 90.0;
// ...
float bar = PI/2;
```

A clever compiler could optimize away the call to **`sin`**

```
void test(double& f) {
  if (f != 90.0)
    f=0;
}

double foo = 90.0;
test(foo);            // may change foo's value
float bar = sin(foo); // -> no optimization here
```

Note: there are compilers that even regard this case.

I.3.2. Advanced Inlining - Cross-Call Optimization

```
inline void test(double& f) {
  if (f != 90.0)
    f=0;
}
double foo = 90.0;
test(foo);
float bar = sin( foo );
```

```
double foo = 90.0;
if (foo != 90.00)
  foo = 0;
float bar = sin( foo );
```

```
double foo = 90.0;
// ..
float bar = sin( foo );
```

```
double foo = 90.0;
float bar = PI/2;
```

Inlining enables optimization.
No inter-procedural analysis required !

I.3.2. Advanced Inlining - Why not inline everything ?

- Inlining may have negative side-effects
  - **Decreased Performance**
    - Suppose a program that (without inlining) never produces cache-faults
    - Inlining may increase code-size; program doesn’t fit into cache no more
    - frequent cache-faults may impact runtime performance severely
  - **Increased compile-time / time for development**
    - Inline functions must appear in header-files
    - Changing the body of an inlined function necessitates recompilation - not just relinking
  - **Debugging becomes more complicated**
    - can’t use single breakpoints to track entry/exit from functions
    - difficult to track variable names across source boundaries
  - **Profiling / Performance Analysis**
    - inlined methods do not appear in program profiles

I.3.2. Advanced Inlining - The Inlining Decision Matrix from Bulka/Mayhew

	Static Size		
Dynamic Frequency	**Large (20+ loc)**	**Medium (5-20 loc)**	**Small (-5 loc)**
Low (the bottom 80% of call frequency)	Do not inline	Do not inline	Consider inlining
Medium (the top 5-20% of call frequency)	Do not inline	Consider rewriting the code to expose its fast path and then inline	Always inline
High (the top 5% of call frequency)	Consider rewriting the code to expose its fast path and then inline	Selectively inline the high frequency static invocation points	Always inline

* taken from Bulka, Mayhew: “Efficient C++” - see references at the end of this section

I.3.2. Advanced Inlining - Conditional Inlining

• delay of inlining

```
// foo.h

#ifnded FOO_H
#define FOO_H

class foo {
  ...
  int bar(int a);
  ...
};

#ifdef INLINE
#include “foo.inl”
#endif

#endif
```

```
// foo.inl

#ifndef INLINE
#define inline
#endif

inline
foo::bar(int a) {
...
};
```

```
// foo.cc

#ifndef INLINE
#include “foo.inl”
#endif


...
```

If **INLINE** is ***not*** defined, the preprocessor removes the inline specifier (it then is defined to be empty)

I.3.3. Reference Counting - Problems with Pointer Data Members I

Consider the following class to represent strings:

```
class String { public:
  String(const char* s=0) : c_string(0) { setString(s); }
  ~String()                { if (c_string) delete [] c_string; }
  int length() const       { return strlen(c_string);          }
  const char* c_str() const { return c_string;                 }
private:
  char* c_string;
  void setString(const char* s) { if (c_string) {
                                    delete [] c_string;
                                    c_string = 0;
                                  }
                                  else if (s) {
                                    c_string=new char [strlen(s)+1];
                                    strcpy(c_string,s);
                                  }
                                }
};
```

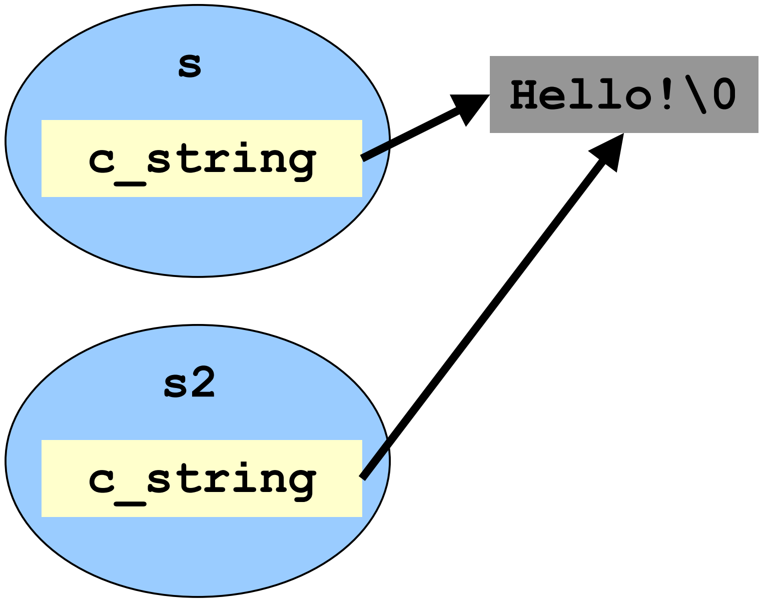
Problem: Automatically generated copy constructor / assignment operator perform bitwise copy !

I.3.3. Reference Counting - Problems with Pointer Data Members II

```
String s("Hello!");

for (int i=0 ; i<10 ; i++) {
   String s2(s);
   ....
}

cout << s.c_str() << endl;
```

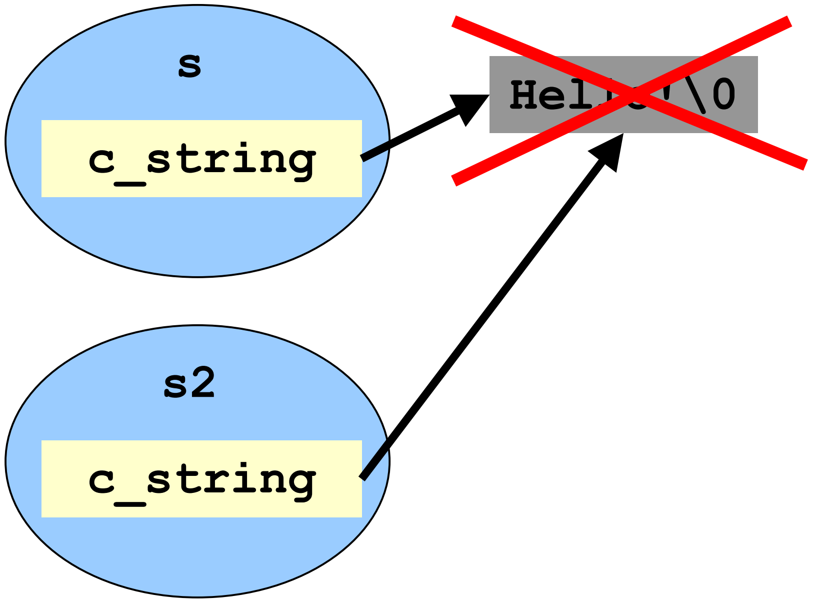
Bitwise copy semantics aplly here !

s2 gets destroyed when leaving scope !



I.3.3. Reference Counting - Problems with Pointer Data Members III

```
String s("Hello!");

for (int i=0 ; i<10 ; i++) {
   String s2(s);
   ....
}

cout << s.c_str() << endl;
```



Bitwise copy semantics aplly here !

s2 gets destroyed when leaving scope !

s.c_string is invalid now !

I.3.3. Reference Counting - Problems with Pointer Data Members IV

Simple solution: whenever a string gets copied (either by a copy constructor or by a call to an assignment operator), we perform a **deep copy**.

```
String& String::operator=(const String& rhs) {
  if (&rhs != this) {
    setString(rhs.c_string);
  }
  return *this;
}
```

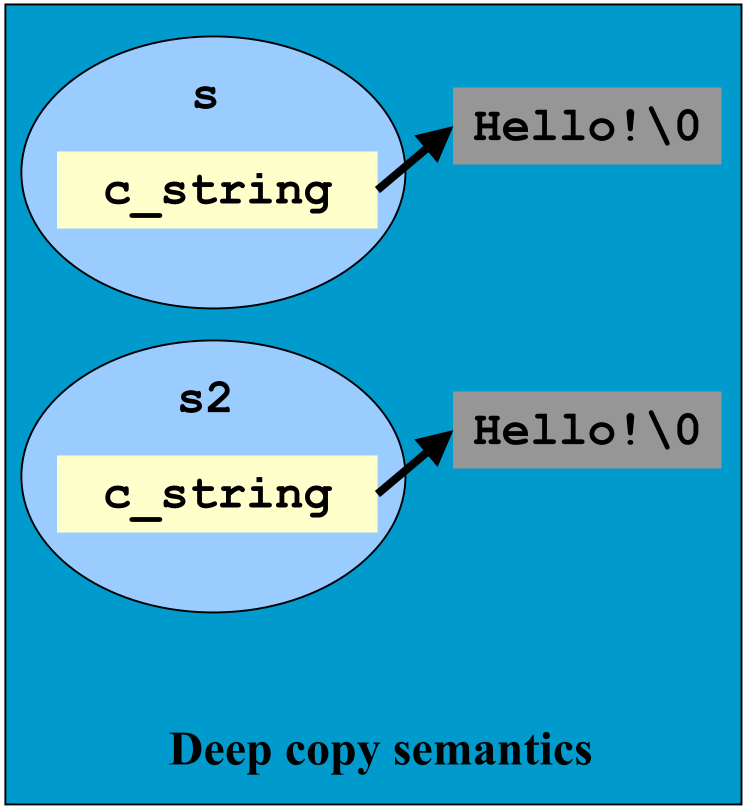
Why perform self-test ?

- **most important**: for safety reasons !
- for speed (avoid unnecessary copy)

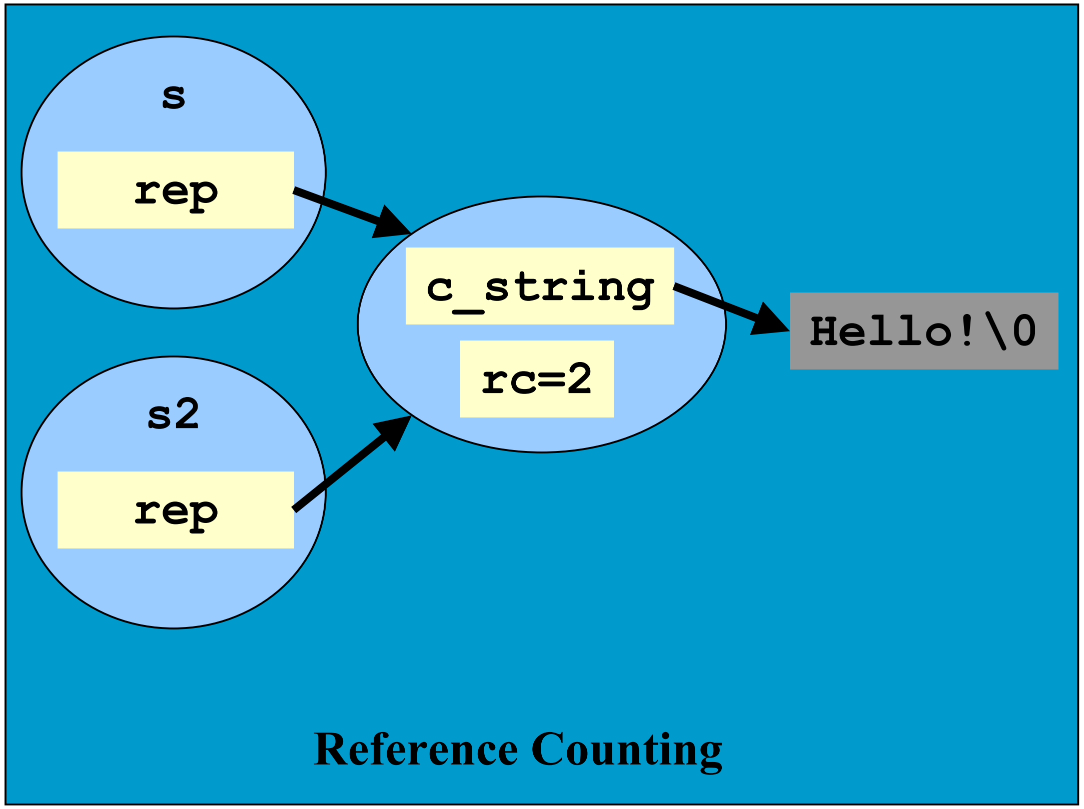
Problem: a) duplicate strings **->** possible waste of memory !
b) copying large/complex objects may become very expensive !

Reference counting helps to avoid duplicate objects.

I.3.3. Reference Counting - Reference Counting I - The Idea



Deep copy semantics



Reference Counting

- store string in separate object
- count how many references to this object exist
- destroy object if it isn't referenced any more

I.3.3. Reference Counting - Reference Counting II - The Handle/Body Class Idiom

Reference counting is a special case of ***handle/body class idiom***

- **Handle class**: user visible class

- **Body class**: helper class for data representation
  - handle class is friend of body class (may access protected/private members)
  - all members of body class are private
  - contains typically only constructor / destructor and necessary data members

```
class StrRep {
  friend class String;
private:
  StrRep(const char* s=0) : c_string(0),rc(0) { setStr(s); }
  ~StrRep() { if (c_string) delete [] c_string; }
  void setStr(const char* s);
  char* c_string;
  int rc;
};
```

I.3.3. Reference Counting - Reference Counting III - The Handle Class I

The handle class **string**

- implements extra intelligence to do the reference counting
- forwards / uses the body class **StrRep**

Private data now pointer to body class **StrRep**:

```
class String {
private:
   StrRep* rep;
```

Default constructor allocates **StringRep** object and sets reference count to **1**

```
public:
  String(const char* s=0) {
    rep = new StrRep(s); rep->rc=1;
  }
```

Copy constructor copies the **rep** object and increments reference counter

```
  String(const String& rhs) {
    rep = rhs.rep; rep->rc++;
  }
```

I.3.3. Reference Counting - Reference Counting III - The Handle Class II

Destructor deletes **rep** object if it is no longer referenced

```
   ~String() {
     if (--rep->rc <=0) delete rep;
   }
```

Access functions “forward” operation to **StrRep** object

```
   const char* c_str() const { return rep->c_string; }
   int length() const        { return strlen(rep->c_stirng; }
```

Other member function that need special attention ....

```
   // Assignment operators
   String& operator=(const String& rhs);
   String& operator=(const char* rhs);

   // Equality test operator
   bool operator==(const Sring& rhs) const;
 };
```

I.3.3. Reference Counting - Reference Counting III - The Handle Class III

The **String** assignment operator

- deletes old **StrRep** object if it is no longer referenced
- copies **StrRep** object and increments reference counter
- returns a reference to allow for daisy chaining

```
String& operator=(const String& rhs) {
  if (rep == rhs.rep)
    return *this;
  if (--rep->rc <=0) delete rep;
  rep = rhs.rep;
  rep->rc++;
  return *this;
}
```

Char* to **String** assignment

```
String& operator=(const char* rhs) {
  if (--rep->rc <=0) delete rep;
  rep = new StrRep(rhs); rep->rc=1;
  return *this;
}
```

I.3.3. Reference Counting - Reference Counting III - The Handle Class IV

The equality test operator

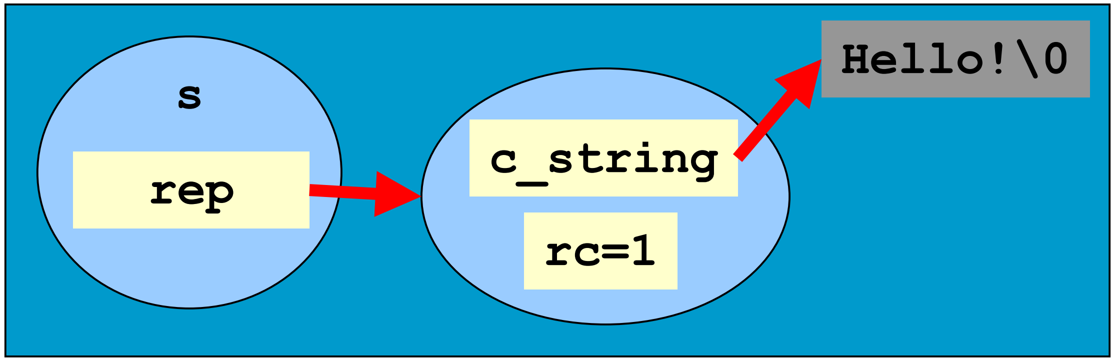
- for speed: compare pointer first
- forwards operation to **StrRep** object

```
 bool operator=(const String& rhs) const {
   if (rep == rhs.rep)
     return true;
   return !strcmp(rhs.rep->str,rep->str);
 }
```

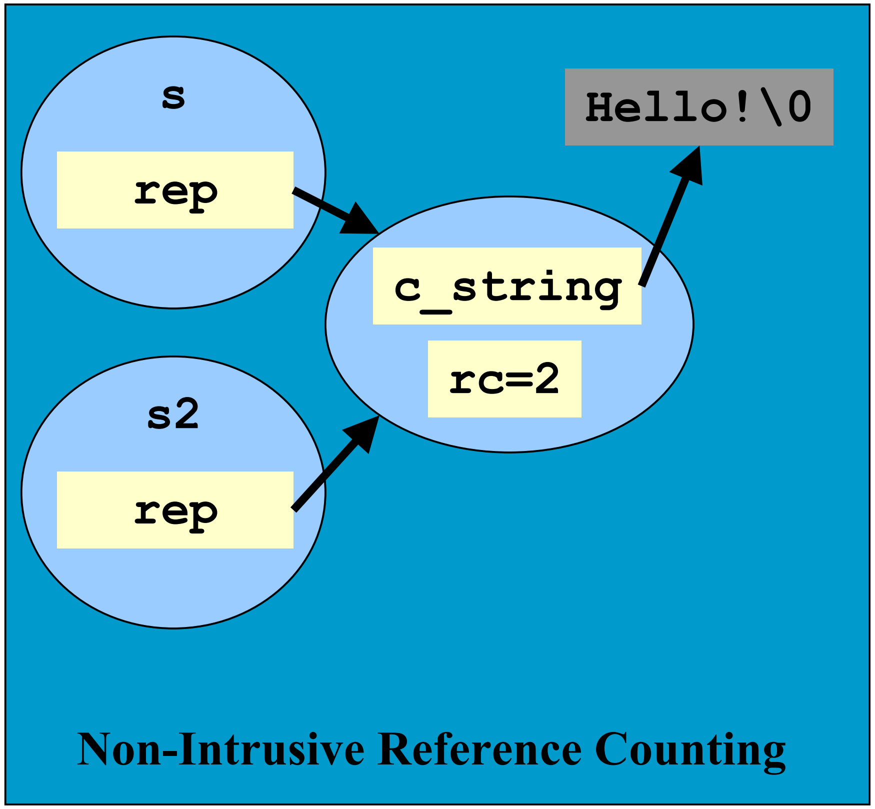
Looks good so far, but ...

... there are two indirections :(

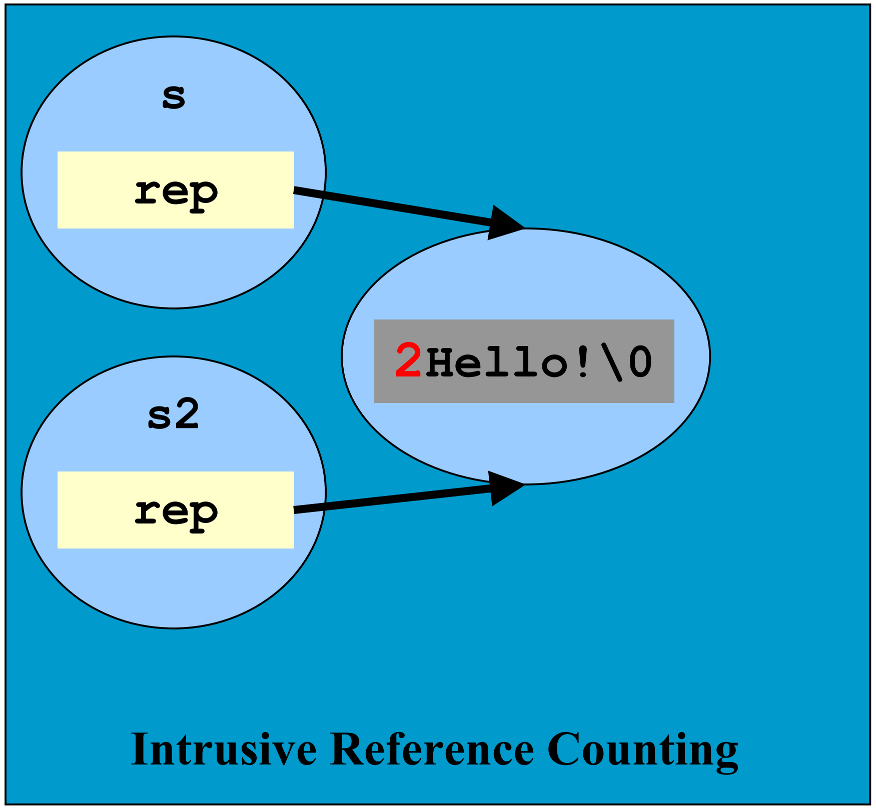
```
String s(“Hello”);
cout << s.c_str() << endl;
```



I.3.3. Reference Counting - Intrusive Reference Counting I



Non-Intrusive Reference Counting



Intrusive Reference Counting

Store reference counter inside of object

I.3.3. Reference Counting - Intrusive Reference Counting II

Make member object aware of reference counting - leave **`String`** class untouched

```
struct StringStorage {
  friend class String;
private:
  struct Data {
    int refCount;
    char c_string[1];
  };
  Data *data;
  void Assign(const char* s) {
    int slen= (strlen(s)+1) * sizeof(char);
    data = static_cast<Data*>( operator new(sizeof(Data)+slen) );
    data->refCount = 1;
  }
  void Release() {
    if (--data->refCount) operator delete(data);
  }

  const char* c_string() const { return data->c_string; }
```

I.3.3. Reference Counting - Intrusive Reference Counting III

```
   StringStorage(const char* s) { Assign(s); }
   ~StringStorage()             { Release(); }

   StringStorage& operator=(const StringStorage& rhs) {
     if (data == rhs.data) return *this;
     Release();
     data = rhs.data;
     ++(data->refCount);
   }

   StringStorage& operator=(const char* rhs) {
     if (rhs==data->c_string) return *this;
     Release();
     Assign(rhs);
   }
 };
```

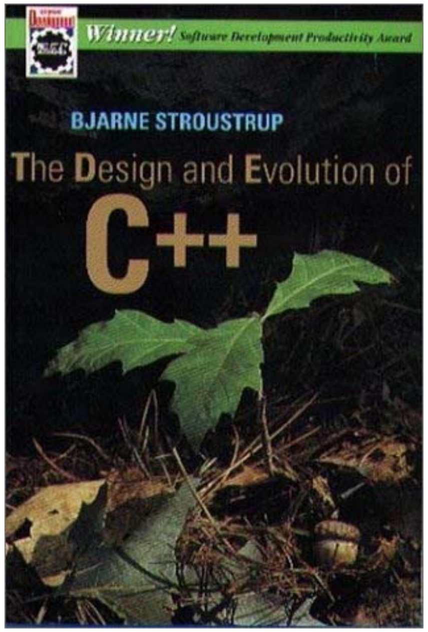
- **`String`** class now stores value of type **`StringStorage`**.
- Bitwise copy semantics do not apply for automatically generated copy constructor / assignment operator !
- Only a single indirection is needed (as for “usual” implementation)

Overview

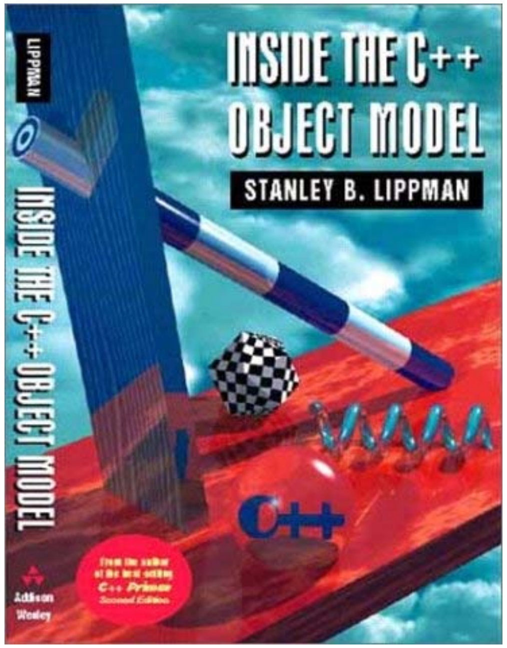
- Classes
- Polymorphism
  - Inheritance
  - Overloading
  - Conversions
  - Templates
- Further Topics
- *References*

I.4. References

- *B. Stroustrup:* **The Design and Evolution of C++,** Addison-Wesley, 1994



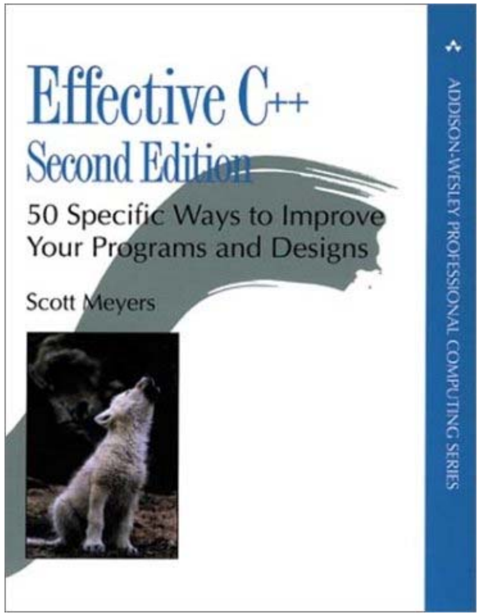
Winner! Software Development Productivity Award
BJARNE STROUSTRUP
The Design and Evolution of
C++

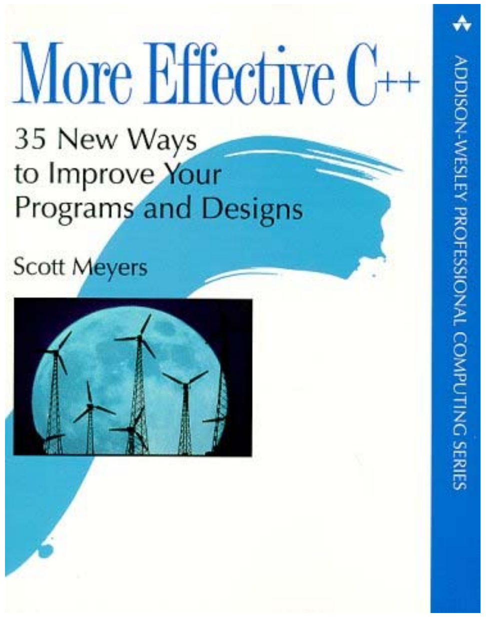


- *Stanley B. Lippmann:* **Inside the C++ Object Model**, Addison-Wesley, 1996

I.4. References

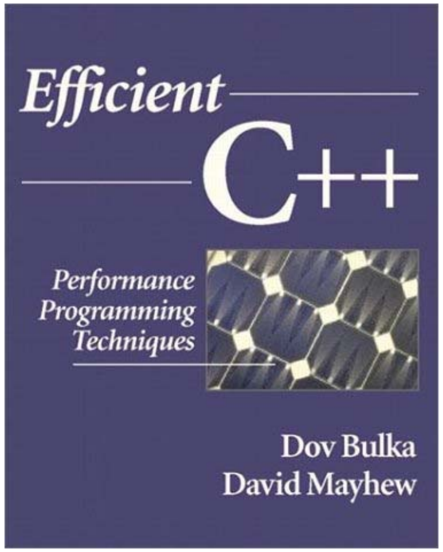
- *S. Meyers:* **Effective C++,** Addison-Wesley, 1997


Effective C++
Second Edition
50 Specific Ways to Improve Your Programs and Designs
Scott Meyers
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

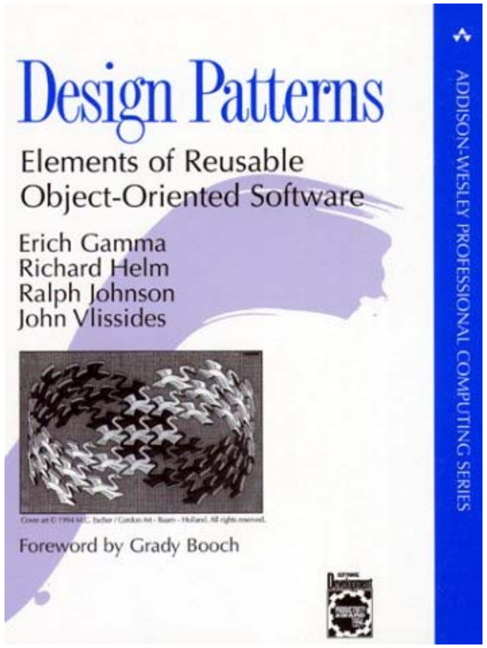

More Effective C++
35 New Ways to Improve Your Programs and Designs
Scott Meyers
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

- *S. Meyers: More* **Effective C++,** Addison-Wesley, 1996

I.4. References



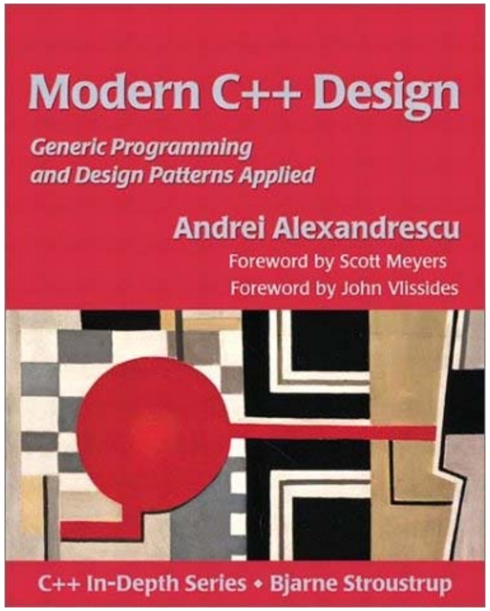
- *D. Bulka / D. Mayhew:* **Efficient C++**, Addison-Wesley, 2000



- *E. Gamma, R. Helm, R. Johnson, J. Vlissides:* **Design Patterns**, Addison-Wesley, 1994

I.4. References

- *A. Alexandrescu:* **Modern C++ Design**
  Addison-Wesley, 2001


Modern C++ Design
Generic Programming
and Design Patterns Applied
Andrei Alexandrescu
Foreword by Scott Meyers
Foreword by John Vlissides
C++ In-Depth Series • Bjarne Stroustrup

- *T. Veldhuizen:* **Techniques for Scientific C++,**
  `http://extreme.indiana.edu/~tveldhui/papers/techniques/`

II. Template Metaprogramming

This section of the course partially is result of joint work with

Krzysztof Czarnecki,
DaimlerChrysler AG
czarnecki@acm.org

&

Ulrich W. Eisenecker,
FH Kaiserslauten
ulrich.eisenecker@t-online.de

Czarnecki/Eisenecker: **Generative Programming**
Addison Wesley, 2000

Jörg Striegnitz
Research Centre Jülich GmbH
j.striegnitz@fz-juelich.de

I always knew that C++ templates were the work of the Devil, and now I'm sure :-)
- Cliff Click

Overview

- *What is template metaprogramming?*
- Metainformation
- Computing values
- Computing types
- Functional Flavor of the Static Level
- Code generation
- Outlook and evaluation
- References

II.1. Metaprogramming / Template Metaprogramming

What is Metaprogramming ?

- **Metaprogramming** is about writing programs that ***represent*** and ***manipulate*** other programs or themselves.
- **Metaprograms** are programs about other programs or themselves, e.g., compilers, interpreters, macros, etc.
- **Metaprograms** can be executed at compile time (**static**) or runtime (**dynamic**).

What is Template Metaprogramming (TMP) ?

- Template metaprograms are executed by the compiler at compile time (**static metaprogramming**).
- They are Representing and manipulating programs (classic metaprogramming).
- They are used to compute static values.

Overview

- What is template metaprogramming?
- *Metainformation*
- Computing values
- Computing types
- Functional Flavor of the Static Level
- Code generation
- Outlook and evaluation
- References

II.2. Metainformation - Overview

Metainformation is information that is available during compile time
- types (including typedefs)
- constants stored in enums
- non-type template parameters

Metainformation on a type could be stored

- in a type itself (as a member)
- using a **traits template**
- using a **traits class**

II.2.1. Metainformation - In a type itself

```
struct Int {
  enum { is_integral = 1, is_exact = 1 };
  // …
};

struct Double {
  enum { is_integral = 0, is_exact = 0 };
  // …
};

template <class IntegralsOnly>
void foo(const IntegralsOnly& I) {
   assert(IntegralsOnly::is_integral == true);
   // …
}
```

II.2.1. Metainformation - In a type itself - Summary

- Information and metainformation located in one place
- Metainformation cannot be extended without invasively modifying the implementation of the type, i.e., no metainformation for basic types like **`double, int`**
- Metainformation for ***various purposes*** is mixed

II.2.2. Metainformation - Traits Templates

Excerpt from **`numeric_limits`**

```
template<class T>
class numeric_limits
{  public:
       static const bool is_specialized = false;
       static T min() throw();
       static T max() throw();
       static const int  digits      = 0;
       static const int  digits10    = 0;
       static const bool is_signed   = false;
       static const bool is_integer  = false;
       static const bool is_exact    = false;
       static const int  radix       = 0;
       static T epsilon() throw();
       static T round_error() throw();
       // ...
```

II.2.2. Metainformation - Traits Templates - Specialization

```
template<>
class numeric_limits<float>
{  public:
       static const bool is_specialized = true;
       inline static float min() throw() { return 1.17549435E-38F; }
       inline static float max() throw() { return 3.40282347E+38F; }
       static const int digits        = 24;
       static const int digits10      =  6;
       static const bool is_signed    = true;
       static const bool is_integer = false;
       static const bool is_exact    = false;
       static const int radix         = 2;
       inline static float epsilon() throw(){
          return 1.19209290E-07F; }
       inline static float round_error() throw() { return 0.5F; }
       // ...
```

Excerpt from the specialization for `numeric_limits<float>`

II.2.2. Metainformation - Traits Templates - Usage

```
cout << numeric_limits<float>::is_integer << endl;
cout << numeric_limits<int>::is_integer << endl;

cout << numeric_limits<complex<float> >::is_specialized << endl;

// “false” implies that numeric_limits was not specialized
// for complex<float>
```

II.2.2. Metainformation - Traits Templates - Summary

- Extending metainformation without modifying the type (**decoupling** of metainformation and type declaration)
- **Modularly** dividing metainformation (**`numeric_limits`**, **`special_metainfo`** …)
- Only one specialization of a traits template per type
- Information and metainformation located in different places

II.2.2. Metainformation - Traits Classes (Configuration Repository)

```
struct SampleConfig {
   typedef double         Element_t;
   typedef unsigned short Size_t;
   enum { size = 5 };
   typedef GenVector<SampleConfig> Vector;
};
```

`SampleConfig` is not a template !

II.2.2. Metainformation - Traits Classes - Importing / Exporting Configurations

• Fixed size vector - configured by **`config`** - traits class.

```
template <class config>
class GenVector {
public:
   typedef config Config_t;             // export configuration
   typedef typename config::Element_t Element_t;
   typedef typename config::Size_t    Size_t;
   GenVector() : size_m(config::size) {
     for (Size_t i=0;i<size();++i) el[i] = 0.0;
   }
   const Size_t& size() const {
     return size_m;
   }
   void element(const Size_t& i,const Element_t& e){ el[i] = e;  }
   const Element_t& element(const Size_T& i) const { return el[i]; }
private:
   Element_t el[config::size];
   const Size_t size_m;
};
```

II.2.2. Metainformation - Traits Classes - Usage

```
typedef SampleConfig::Vector Vector;
Vector v;
v.element(2,42.1);
cout << v.size();

typedef Vector::Size_t Size_t;
for (Size_t i=0; i<v.size() ; ++i)
  cout << '\t' << v.element(i);
cout << endl;
```

`SampleConfig::Vector` is the vector type that is provided by the ***configuration repository*** `SampleConfig`.

II.2.3. Metainformation - Structured Configuration Repository

A configuration repository can provide configurations for more than one type:

```
struct SampleConfig {
  struct ForVector {
    typedef int Element_t;
     enum { size = 5 };
  };
  struct ForMatrix {
    typedef double Element_t;
    enum { rows = 5, cols = 5 };
  };
  typedef GenVector<SampleConfig> Vector;
  typedef GenMatrix<SampleConfig> Matrix;
};
```

II.2.3. Metainformation - Structured Configuration Repository

```
template <class config>
class GenVector {
public:
  typedef config Config;
  typedef typename Config::ForVector MyConfig;
  typedef typename MyConfig::Element_t Element_t;
   // ...
};

template <class config>
class GenMatrix {
public:
  typedef config Config;
  typedef typename Config::ForMatrix MyConfig;
  typedef typename MyConfig::Element_t Element_t;
   // ...
};
```

II.2.3. Metainformation - Structured Configuration Repository - Usage

```
typedef SampleConfig::Vector Vector;
typedef SampleConfig::Matrix Matrix;
Vector v;
Matrix m;
// ...
```

II.2.4. Metainformation - Computing Configuration Repositories

```
struct Precise {
  typedef long double Element_t;
};

struct Compact {
  typedef float Element_t;
};
// Configuration Parameters
template <int Size = 10, class Precision = Compact>
struct ComputeConfig {  // Configuration Repository
  struct Config {
    typedef typename Precision::Element_t Element_t;
    enum { size = Size };
    typedef Vector<Config> Vector;
  };
};
```

II.2.4. Metainformation - Computing Configuration Repositories

```
template <class config>
class GenVector {
public:
  typedef config Config; // export config
  typedef typename Config::Element_t Element_t;
      // ...
};

typedef ComputeConfig<5,Precise>::Config SampleConfig;
typedef SampleConfig::Vector Vector;
Vector v;
// ...
```

Compute Configuration Repository

II.2.5. Metainformation - Traits Classes - Summary

- Parameterizing components with different traits classes in a flexible way
- Traits classes containing configuration information for a set of components (configuration repository)
- Structured configuration repositories
- Computing configuration repositories
- Information and metainformation located in different places
- Configuration repositories can get large

Overview

- What is template metaprogramming?
- Metainformation
- *Computing values*
- Computing types
- Functional Flavor of the Static Level
- Code generation
- Outlook and evaluation
- References

II.3. Computing Values - Overview

- Computing constants
- Simple template metafunctions (**TMF**)
- TMF calling other TMF
- TMF with several statements
- TMF as TMF parameters

II.3.1. Computing Values - Computing Constants - Advantages

```
const double PI = 22.0/7.0;
const double PI_SQUARED = PI*PI;
```

- More descriptive (higher intentionality)
- Implementation (the computation formula) easy to modify
- Depended on ***meaningful names***

II.3.1. Computing Values - Computing Constants - Examples

```
const int C1 = 40320;
```

- not descriptive
- hard to maintain
- value fixed at compile-time

```
const int C2 = 1*2*3*4*5*6*7*8;
```

- more descriptive
- still hard to maintain
- computed at compile-time

```
int factorial(int n) {
  return (n==0) ? 1 : n*factorial(n-1);
}

const int C3 = factorial(8);
```

- very descriptive
- easy to maintain
- computed at runtime

II.3.2. Computing Values - Template Metafunction

```
fac 0 = 1
fac n = n * fac (n-1)
```

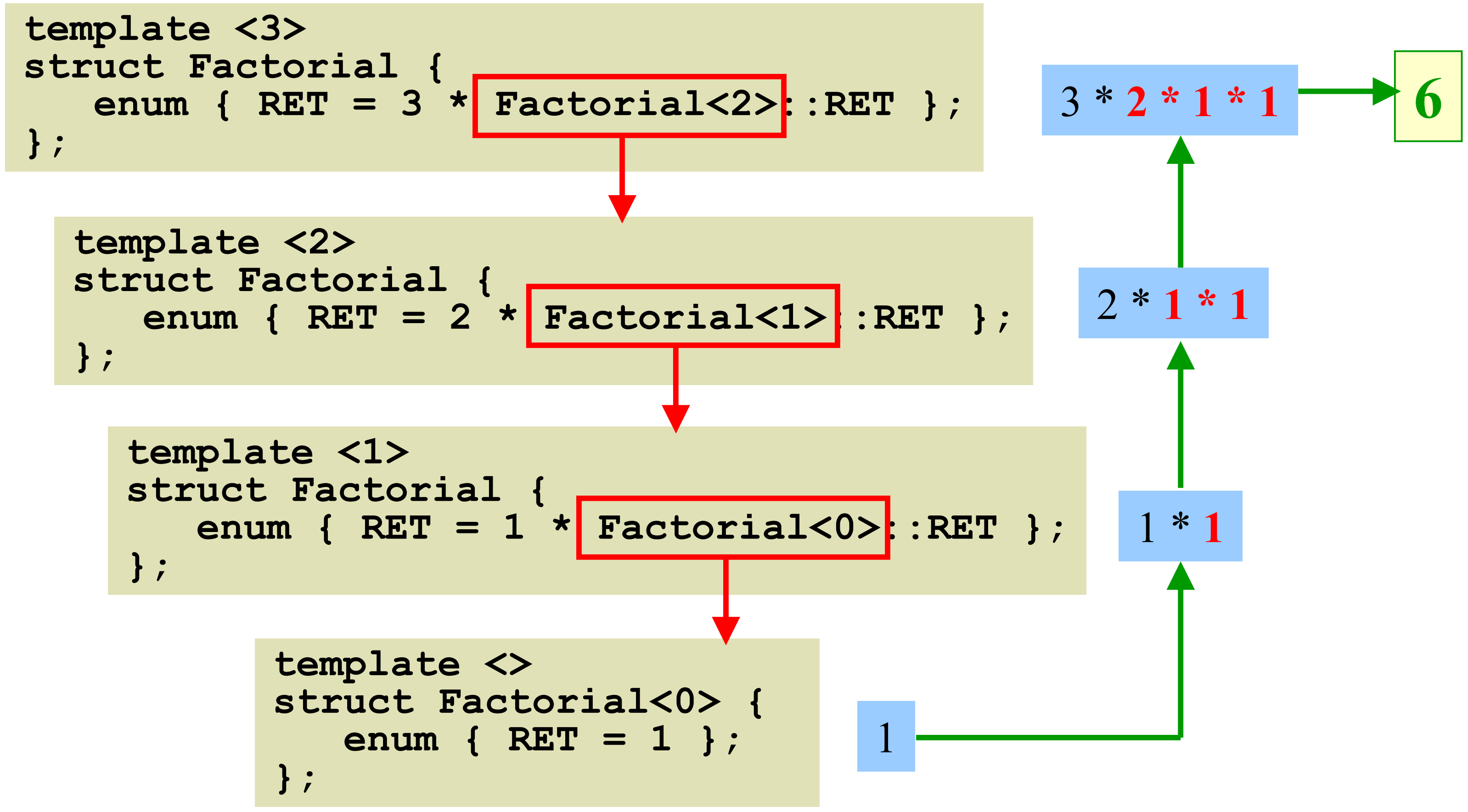
```
template <int n>
struct Factorial {
   enum { RET = n * Factorial<n-1>::RET };
};

template <>                    // Specialization to end the recursion
struct Factorial<0> {
   enum { RET = 1 };
};
```

```
const int C4 = Factorial<8>::RET;
```

- very descriptive
- easy to maintain
- computed at compile-time

```
cout << Factorial<8>::RET << endl; //  cout << 40320 << endl;
```

II.3.2. Computing Values - Template Metafunctions - Example



```
template <3>
struct Factorial {
    enum { RET = 3 * Factorial<2>::RET };
};
```

```
template <2>
struct Factorial {
    enum { RET = 2 * Factorial<1>::RET };
};
```

```
template <1>
struct Factorial {
    enum { RET = 1 * Factorial<0>::RET };
};
```

```
template <>
struct Factorial<0> {
    enum { RET = 1 };
};
```

Class templates can be used as **compile-time functions**

II.3.2. Computing Values - TMF calling other TMF

- Take k from n elements
  - without putting elements back
  - without taking order into account

(Combinations without repetitions)

$$C(k,n) = \frac{n!}{k! \cdot (n-k)!}$$

II.3.2. Computing Values - TMF calling other TMF

```
Combinations k n = fac n / ( fac (k) * fac (n-k) )
```

```
template<int k, int n>
struct Combinations  {
  enum { RET = Factorial<n>::RET /
               (Factorial<k>::RET * Factorial<n-k>::RET)
       };
};
```

```
cout << Combinations<2,4>::RET << endl;
```

II.3.2. Computing Values - TMF with several statements

- A TMF may contain multiple statements

```
template<int k, int n>
class Combinations {
  enum {  num   = Factorial<n>::RET,   // statement 1
          denom = Factorial<k>::RET *  // statement 2
          Factorial<n-k>::RET
       };
public:
  enum { RET = num / denom };          // statement 3
};
```

II.3.2. Computing Values - TMF as TMF Parameter (Template Template Para.)

- Consider **accumulate** function (like in the STL):

```
accumulate(n,f) := f(0) + f(1) + ... +f(n)
```

```
// Passing a template as a template parameter
template<int n, template<int> class F>
struct Accumulate {
  enum { RET = Accumulate<n-1,F>::RET + F<n>::RET };
};

template<template<int> class F>
struct Accumulate<0,F> {
  enum { RET = F<0>::RET };
};
```

II.3.2. Computing Values - TMF as TMF argument

```
template<int n, template<int> class F>
struct Accumulate {
  enum { RET = Accumulate<n-1,F>::RET + F<n>::RET };
};

template<template<int> class F>
struct Accumulate<0,F> {
  enum { RET = F<0>::RET };
};
```

```
template<int n>
struct Square {
  enum { RET = n*n };
};
// ...
cout << Accumulate<3,Square>::RET << endl;
```

```
Accumulate<2,F>::RET + Square<3>::RET

Accumulate<1,F>::RET + Square<2>::RET + Square<3>::RET

Accumulate<0,F>::RET + Square<1>::RET + Square<2>::RET + Square<3>::RET

Square<0>::RET + Square<1>::RET + Square<2>::RET + Square<3>::RET

0*0 + 1*1 + 2*2 + 3*3          cout << 14 << endl
```

II.3.2. Computing Values - TMF as TMF Parameter (Member Templates)

- **Problem:** only a few compiler support template template parameter
- **Solution:** Use **member templates** instead:

```
struct Square {
  template<int n>
  struct Apply {
    enum { RET = n*n };
  };
};

template<int n, class F>
struct Accumulate {
  enum { RET = Accumulate<n-1,F>::RET + F::template Apply<n>::RET };
};

template<class F>
struct Accumulate<0,F> {
  enum { RET = F::template Apply<0>::RET };
};
```

II.3.3. Computing Values - Summary

- Computation at compile time, thus
  - better performance
  - smaller object code
- Descriptive and easy to maintain representations of constants
- Full spectrum as for dynamic function calls
  - Functions calling other functions
  - Functions as function parameters
- Recursion as the only iteration mechanism
- Unusual syntax
- Longer compilation times

Overview

- What is template metaprogramming?
- Metainformation
- Computing values
- *Computing types*
- Functional Flavor of the Static Level
- Code generation
- Outlook and evaluation
- References

II.4. Computing Types - Overview

- Simple selection (**`If`**)
- Multiple selection (**`switch`** / **`case`**)
- Decision tables

II.4.1. Computing Types - Simple Selection - Application Example

Consider a matrix library.

- `Matrix<int> + Matrix<float>` ⌫ `Matrix<float>`
- **Solution:** TMF selecting the conceptually larger type
- The **necessary metainformation is available** (`numeric_limits<>::max_exponent10`, `numeric_limits<>::digits`)
- **Required:** A ***simple selection*** of one from two types based on a Boolean expression

II.4.1. Computing Types - Simple Selection

```
template<bool condition, class THEN, class ELSE>
struct IF {                        // condition = true
   typedef THEN RET;
};

// partial specialization
template<class THEN, class ELSE>
struct IF<false, THEN, ELSE> { // condition = false
   typedef ELSE RET;
};

IF<(1+2>4), int, float>::RET i;
```

$$\begin{tikzcd} (1+2>4) \arrow[d] \\ 3>4 \arrow[d] \\ \text{false} \end{tikzcd}$$

`IF<false, int, float>::RET` yields `float`

II.4.1. Computing Types - Simple Selection

```
template<class A, class B>
struct Promote {
  enum {
    cond = (numeric_limits<A>::max_exponent10 <
            numeric_limits<B>::max_exponent10) ||
           ( numeric_limits<A>::max_exponent10 ==
             numeric_limits<B>::max_exponent10 &&
             numeric_limits<A>::digits <
             numeric_limits<B>::digits )
   };
  typedef typename IF<cond,B,A>::RET RET; // A<B ? A : B;
};
```

II.4.1. Computing Types - Simple Selection - Example

```
template <class T1,class T2>
Matrix<Promote<T1,T2>::RET> operator+(Matrix<T1>& m1,Matrix<T2>& m2){
  Matrix<Promote<T1,T2>::RET> result;
  // ... Perform computation
  return result;
}

//…
typedef int    Type1;
typedef double Type2;
Matrix<Type1> m;
Matrix<Type2> n;
Matrix<Promote<Type1,Type2>::RET> r = m + n; // Matrix<double>

// or you may wish save the result of Promote for future use ...
typedef Promote<Type1,Type2>::RET RET;
Matrix<RET> r = m + n; // Matrix<double>
```

II.4.1. Computing Types - Simple Selection - Alternative Implementation

• Some compilers do not allow for partial specialization (e.g. Microsoft's Visual C++)
• **Idea:** Regard **`if`** as a function that selects one value out of a tuple.

```
// class that offers a metafunction that returns the first
// element of a tuple
struct SelectThen  {
   template<class ThenType, class ElseType>
   struct Result {
      typedef ThenType RET;
   };
};

// class that offers a metafunction that returns the second
// element of a tuple
struct SelectElse {
   template<class ThenType, class ElseType>
   struct Result {
      typedef ElseType RET;
   };
};
```

II.4.1. Computing Types - Simple Selection - Alternative Implementation

```
template<bool condition> // Choosing a selecting class
struct ChooseSelector
{ typedef SelectThen RET; };

template<>
struct ChooseSelector<false>  // Full specialization !!
{ typedef SelectElse RET; };

template<bool condition, class ThenType, class ElseType>
struct IF {
  typedef typename ChooseSelector<condition>::RET Selector;
  typedef typename Selector::template Result<ThenType,ElseType>::RET
           RET;
};
```

II.4.1. Computing Types - Simple Selection - Summary

- **`IF<>::RET`** returns a type
- Used like a “normal” **`if`**
- **`IF`** and its evaluation are easy to save
  - **`typedef IF< 1!=2, A, B> Result;`**
  - **`typedef IF< 1!=2, A, B>::RET Result;`**
- Type-dependent ***selection of code*** or member access
  - **`IF< 1!=2, A, B>::RET::some_static_function();`**
  - **`int result = IF< 1!=2, A, B>::RET::some_int;`**

II.4.2. Computing Types - Multiple Selection

- Customization of a automatic teller machine (ATM)
- Messages in a local language depending on the country of installation
- One ATM **contains only the code for the needed language**
- **Configuration at compile time**

II.4.2. Computing Types - Multiple Selection

```
struct SpracheDeutsch {
  static const char* kreditUeberzogen() {
    return "Kreditrahmen ueberschritten";
  }
  static const char* limitUeberschritten() {
    return "Hoechstbetrag ueberschritten";
  }
  static const char* zuwenigGeld() {
    return "Unzureichender Geldvorrat";
  }
  static const char* sonstigeStoerung() {
     return "Funktionsstoerung";
  }
};

// SpracheEnglisch, SprachePolnisch
```

II.4.2. Computing Types - Multiple Selection

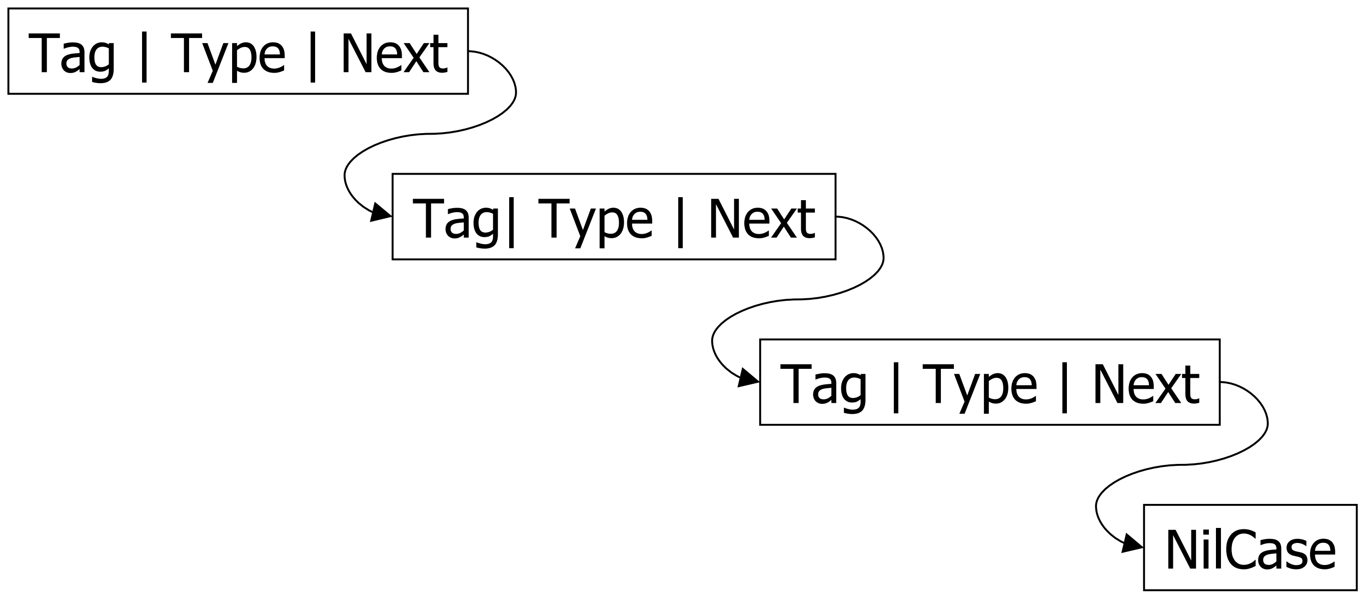
```
template <class Landessprache> // Parameterized Polymorphism !
class Bankautomat : protected Landessprache {
private:
  typedef Bankautomat<Landessprache> self;
public:
  // ...
  void auszahlen(const double& betrag) {
    if (kreditrahmenUeberzogen(betrag)) {
      cout << self::kreditUeberzogen() << endl;
      return;
     }
     if (betrag > limit()) {
       cout << self::limitUeberschritten() << endl;
       return;
     }
// ...
```

II.4.2. Computing Types - Multiple Selection

```
enum Land { Deutschland, England, Oesterreich, Polen, USA };
// …
LOKALISIERE_BANKAUTOMAT<Deutschland>::RET bankautomat;
teste(bankautomat);
```

```
template <Land land>
struct LOKALISIERE_BANKAUTOMAT {
  typedef typename
          SWITCH<land,
                 CASE<Deutschland, SpracheDeutsch,
                 CASE<England,     SpracheEnglisch,
                 CASE<Oesterreich, SpracheDeutsch,
                 CASE<Polen,       SprachePolnisch,
                 CASE<USA,         SpracheEnglisch
                > > > > > >::RET Sprache;
  typedef Bankautomat<Sprache> RET;
};
```

II.4.2. Computing Types - Multiple Selection - Implementation



II.4.2. Computing Types - Multiple Selection - Implementation

```
const int DEFAULT = ~(~0u >> 1); // plattform-independent: smallest
                                 // integer value

struct NilCase {};

template <int TAG, class TYPE, class NEXT = NilCase>
struct CASE {
  enum { tag = TAG };
  typedef TYPE Type;
  typedef NEXT Next;
};
```

II.4.2. Computing Types - Multiple Selection - Implementation

```
template<int tag, class Case> // Solution with partial specialization
struct SWITCH {
   typedef typename Case::Next NextCase;
   enum     { caseTag     = Case::tag,
              found       = (caseTag == tag || caseTag == DEFAULT)
            };
   typedef typename
            IF<found,
                 typename Case::Type, // return head of list
                 typename SWITCH<tag, NextCase>::RET // search in tail
               >::RET RET;
};


template<int tag>
struct SWITCH<tag, NilCase> {
   typedef NilCase RET;
};
```

II.4.2. Computing Types - Multiple Selection - Summary

- Usage like a “normal” switch; different implementations possible
  - **`DEFAULT<`**…**`>`** instead of **`CASE<DEFAULT,`** …**`>`**
  - **`DEFAULT<`**…**`>`** can stand at an arbitrary place
- Implementation with member templates possible
- Special checks
  - Detecting multiple **`DEFAULT`** branches
  - Detecting multiple tags

II.4.3. Computing Types - Dependency Tables

- Customizing output language based on country and state
- One ATM contains only the code for the needed language
- Configuration at compile time

II.4.3. Computing Types - Dependency Tables

Land	Bundesland	**Sprache**
Deutschland	*	`BankautomatDeutsch`
England	*	`BankautomatEnglisch`
Österreich	*	`BankautomatDeutsch`
Polen	*	`BankautomatPolnisch`
USA	*	`BankautomatEnglisch`
Kanada	Québec	`BankautomatFranzösisch`
Kanada	*	`BankautomatEnglisch`
Schweiz	Jura	`BankautomatFranzösisch`
Schweiz	Wallis	`BankautomatFranzösisch`
Schweiz	Genf	`BankautomatFranzösisch`
Schweiz	Freiburg	`BankautomatFranzösisch`
Schweiz	Waadt	`BankautomatFranzösisch`
Schweiz	Tessin	`BankautomatItalienisch`
Schweiz	*	`BankautomatDeutsch`
*	*	`BankautomatEnglisch`

II.4.3. Computing Types - Dependency Tables - Example

- `land` and `bundesland` are input-types

```
typedef typename EVALTABLE
//**********************************************************************
<      CELL<       land , CELL<  bundesland                           > >
//----------------------------------------------------------------------
, ROW< CELL<Deutschland , CELL<  anyValue    , SpracheDeutsch          > >
, ROW< CELL<   England  , CELL<  anyValue    , SpracheEnglisch         > >
, ROW< CELL< Oesterreich, CELL<  anyValue    , SpracheDeutsch          > >
, ROW< CELL<     Polen  , CELL<  anyValue    , SprachePolnisch         > >
, ROW< CELL<      USA   , CELL<  anyValue    , SpracheEnglisch         > >
, ROW< CELL<   Kanada   , CELL<   Quebec     , SpracheFranzoesisch   > >
, ROW< CELL<   Kanada   , CELL<  anyValue    , SpracheEnglisch         > >
, ROW< CELL<   Schweiz  , CELL<     Jura     , SpracheFranzoesisch   > >
, ROW< CELL<   Schweiz  , CELL<    Wallis    , SpracheFranzoesisch   > >
, ROW< CELL<   Schweiz  , CELL<      Genf    , SpracheFranzoesisch   > >
, ROW< CELL<   Schweiz  , CELL<   Freiburg   , SpracheFranzoesisch   > >
, ROW< CELL<   Schweiz  , CELL<    Waadt     , SpracheFranzoesisch   > >
, ROW< CELL<   Schweiz  , CELL<    Tessin    , SpracheItalienisch    > >
, ROW< CELL<   Schweiz  , CELL<   Zuerich    , SpracheDeutsch          > >
, ROW< CELL< anyValue   , CELL<  anyValue    , SpracheEnglisch         > >
//**********************************************************************
> > > > > > > > > > > > > > > >::RET Sprachanpassung;
```

Yes, that's C++ code !

II.4.3. Computing Types - Dependency Tables - Usage

```
enum Land {    Deutschland,  // etc. };

enum Bundesland  {
   // Deutschland
   BadenWuerttemberg,  // etc.
};

template <Land land, Bundesland bundesland>
struct LOKALISIERE_BANKAUTOMAT {
   typedef typename EVAL_TABLE<
      // ... See previous slide
   >::RET Sprachanpassung;
   typedef Bankautomat<Sprachanpassung> RET;
};

LOKALISIERE_BANKAUTOMAT<Schweiz,Wallis>::RET bankautomat;
teste(bankautomat);
```

II.4.3. Computing Types - Dependency Tables - Summary

- Implementation somewhat more complex (see Knaupp, Eisenecker, Czarnecki: Mit Tabellen zur Entscheidung, in OBJEKTspektrum 5/99)

- Extended dependency tables
  - Evaluation in the top-down direction
  - joker (**`anyValue`**)

- Returning multiple values
  - Chaining results in lists
  - Configuration repositories

Overview

- What is template metaprogramming?
- Metainformation
- Computing values
- Computing types
- *Functional Flavor of the Static Level*
- Code generation
- Outlook and evaluation
- References

II.5. Functional Flavor of the Static Level

- **Static C++ code is functional**, writing static C++ code is like programming in a functional language.

- Functional programming languages
  - – have no variables
  - – no assignment
  - – no iterative constructs
  - – are based on the concept of mathematical functions (lambda calculus)
  - – use recursions instead of loops

- Class templates as functions
  - compile-time functions
  - take types/integers as arguments and return types or integers
  - template instantiation ⌦ function application, however, when writing static code we usually separate return value from template itself (**`Factorial<5>::RET`**)

- Integers and types as data
  - Static code operates on integers and types as data
  - We will later see how to represent complex data (e.g. lists, trees)

II.5. Functional Flavor of the Static Level

- Symbolic names instead of variables
    - variables are `typedef` names and integral constants
    - they are initialized once - their value cannot be changed
    - use symbolic names rather than true variables
- Constant initialization and `typedef` - Statement instead of assignment
    - use `enum` declaration if you need an integer value at the static level
    - static constant members are considered as misfeature by Stroustrup
    - use `typedef` to introduce member types
- Template recursion instead of loops
    - as in functional programming there is no loop statement
    - it is possible to mimic `FOR`, `DO`, `WHILE`
    - take care of infinite recursion
    - recursion may be limited by number of “pending instantiations”

Overview

- What is template metaprogramming?
- Metainformation
- Computing values
- Computing types
- Functional Flavor of the Static Level
- *Code generation*
- Outlook and evaluation
- References

II.6. Code Generation - Overview

- Code generating TMFs
- Simple Code Selection
- DO
- WHILE
- FOR

II.6.1. Code Generation - Practical Example

- Many language dependent attributes for ATM exits
- Each possible instance has to be tested !

II.6.1. Code Generation - Practical Example - Explicit Coding

```
{   LOKALISIERE_BANKAUTOMAT<Deutschland>::RET bankautomat;
    teste(bankautomat);
}
{   LOKALISIERE_BANKAUTOMAT<England>::RET bankautomat;
    teste(bankautomat);
}
// ...
```

- redundant code
- error-prone
- hard to maintain

II.6.1. Code Generation - Practical Example

```
enum Land { Deutschland, England, Oesterreich, Polen, USA, STOP };
// LOKALISIERE_BANKAUTOMAT reamins unchanged
template <Land land = Deutschland>
struct test_all {
  static void execute() {
    LOKALISIERE_BANKAUTOMAT<land>::RET bankautomat;
    teste(bankautomat);
    typedef test_all<Land(land + 1)> naechsterTest;
    naechsterTest::execute();
  }
};

// explicit spcialization to terminate recursion
template <> struct test_all<STOP> {
  static void execute() {}
};
// ...
test_all<>::execute();
```

II.6.2. Code Generation - Simple Code Selection

```
struct algoA {
  static void execute() {
    cout << “Hello A” << endl;
  }
};

struct algoB {
  static void execute() {
    cout << “Hello B” << endl;
  }
};


IF<(1<2), algoA, algoB>::RET::execute(); // cout<<“Hello A”<<endl;
```

• `execute` is a **static inline function** - compiler can optimize away any overhead
• code selection criteria may depend on very complex (static) computations
• Note the difference to preprocessor approaches - metaprograms allow for much more control (e.g. preprocessor directives may not contain C++ static data)

II.6.3. Code Generation - Recursive Code Expansion

- Template Metaprograms can be used to expand code recursively
- optimize code (e.g. loop unrolling), generate test code

```
inline int power(const int& m,int n) {
  int r=1;
  for ( ;n>0; --n) r*=m;
  return r;
}
int m,k;
cin >> m;
k = power(m,3);
```

- The **exponent** is known at compile-time !
- unroll loop to optimize code:

```
inline int power3(int& m) {
  in r=1; r*=m ; r*=m ; r*=m:
  return r;
}
```

- Although done by compilers automatically, TMFs give you more control !

II.6.3. Code Generation - Recursive Code Expansion - Example

```
template <int n>
inline
int power(const int& m) {
   return power<n-1>(m) * m;
}

template<>
inline
int power<1>(const int& m) {
   return m;
}

template<>
inline
int power<0>(const int& m) {
   return 1;
}
```

$$\begin{tikzcd}
\texttt{cout<< power<3>(m) << endl;} \arrow[d] \\
\texttt{cout<< power<2>(m) * m << endl;} \arrow[d] \\
\texttt{cout<< power<1>(m) * m * m << endl;} \arrow[d] \\
\texttt{cout<< 1 * m * m * m << endl;}
\end{tikzcd}$$

- different calling syntax !
- unrolling not limited

• In case your compiler doesn't support explicit function template specialization: use class templates

II.6.3. Code Generation - Recursive Code Expansion - Optimize Vector Operations

```
inline
void add_vec(int size, const double* a,const double *b,double *c) {
  while (size--) *c++ = *a++ + *b++;
}

double a[3] = { 1.1, 2.2, 3.3 };
double b[3] = { -1.0, -2.0, -3-0 };
double c[3];

add_vec(3, a,b,c); // size of vector known during compile-time !
```

```
template <int size>
inline void add_vec(const double* a,const double* b,double *c) {
  *c = *a + *b;
  add_vec<size-1>(a+1,b+1,c+1); // recursive function call
}

template <>
inline void add_vec<0>(const double* a,const double* b,double *c) {}
```

II.6.3. Code Generation - Recursive Code Expansion - Vector Example

```
add_vec<3>(a,b,c);
```

↓

```
*c = *a + *b;
add_vec<2>(a+1,b+1,c+1);
```

↓

```
*c     =      *a +     *b;
*(c+1) = *(a+1) + *(b+1);
add_vec<1>(a+1+1,b+1+1,c+1+1);
```

↓

```
*c     =      *a +     *b;
*(c+1) = *(a+1) + *(b+1);
*(c+2) = *(a+2) + *(b+2);
add_vec<0>(a+1+1+1,b+1+1+1,c+1+1+1);
```

↓

```
*c     =      *a +     *b;
*(c+1) = *(a+1) + *(b+1);
*(c+2) = *(a+2) + *(b+2);
```

II.6.3. Code Generation - Recursive Code Expansion - Limiting Depth of Recursion 1

```
struct OPT_ENVIRON { enum { unroll_depth=2 }; }; // Traits class !

template<int n> inline int power(const int& m)  {
  return IF< n <= OPT_ENVIRON::unroll_depth,
             unroll_power<n>,
             normal_power<n>
           >::RET::exec(m); // code selection !
}

// needs to be defined before power !
template <int n> struct unroll_power {
  static inline int exec(const int& m)  {
    return unroll_power<n-1>::exec(m)  * m;
};

template <> struct unroll_power<1> {
  static inline int exec(const int& m)  { return m; }
};

template <> struct unroll_power<0> {
  static inline int exec(const int& m)  { return 1; }
};
```

II.6.3. Code Generation - Recursive Code Expansion - Limiting Depth of Recursion 2

```
template <int n> struct normal_power {
  static inline int exec(const int& m) {
    int r=1;
    for (int i=0; i< N ; i+=OPT_ENVIRON::unroll_depth)
      r*=unroll_power<unroll_depth>::exec(m);
    for (int i=0 ; i < N %  OPT_ENVIRON::unroll_depth ; i++)
      r*=m;
    return r;
  }
}
```

• calling syntax still different !

II.6.3. Code Generation - Recursive Code Expansion - Compile time / Runtime

- `n/m` known at compile time:

```
template <int m,int n> struct Power {
   enum { RET = Power<m,n-1>::RET * m  };
};

template <int m> struct Power<m,0> {
   enum { RET = 1 };
};
```

- **`n`** known at compile time: use **`power<n>`**
- **`n`** and **`m`** are not known until runtime: use **`power`** function

- How to establish a uniform interface / a single **`power`** function

➔ Move information into Type of **`m`** / **`n`**

```
template <int n> StaticInt {
   enum { RET = n };
   operator const int() const { return n; }
};
```

II.6.3. Code Generation - Recursive Code Expansion - Offline Partial Evaluation

```
// Power<int,int>(), power<int>(int), power(int,int), as before

template <int m,int n>
inline StaticInt<Power<m,n>::RET> power(const StaticInt<m>&,
                                        const StaticInt<n>&) {
  return StaticInt<Power<m,n>::RET>;
}

template <int n>
inline int power(const int& m,const StaticInt<n>&) {
  return power<n>(m);
}
StaticInt<2> c2;
StaticInt<3> c3;
cout << power(2,3)   << endl;       // uses loop
cout << power(c2,3)  << endl;       // uses loop
cout << power(2,c3)  << endl;       // cout << 2*2*2 << endl;
cout << power(c2,c3) << endl;       // cout << 8     << endl;
cout << power(c2,power(c2,c3)) << endl; // cout << 256 << endl;
```

II.6.4. Code Generation - Imperative Style - Do & WHILE

```
enum Land { Deutschland, England, Oesterreich, Polen, USA };
// No need for STOP! LOKALISIERE_BANKAUTOMAT remains unchanged

template <Land land = Deutschland>
struct Statement {
  enum { land_ = land };
  static void exec() {
    LOKALISIERE_BANKAUTOMAT<land>::RET bankautomat;
    teste(bankautomat);
  }
  typedef Statement<Land(land + 1)> Next;
};

struct Condition {
  template <class STATEMENT>
  struct Code {
    enum { RET = Land(STATEMENT::land_) <= USA };
  };
};
// ...
```

```
DO<Statement<>,Condition>::exec();
WHILE<Condition,Statement<> >::exec();
```

II.6.4. Code Generation - Imperative Style - Implementation of DO

```
struct Stop {
  static inline void exec() {};
};

template <class Statement, class Condition>
struct DO {
  typedef typename Statement::Next NextStatement;
  static void exec() {
    Statement::exec();                // Statement.exec
    IF <Condition::template Code<NextStatement>::RET,
        DO<NextStatement,Condition>,
        Stop                          // if (Condition)
      >::RET::exec();                 //     DO(NextStatement,Condition)
   }                                  //  else
};                                    //     Stop.exec
```

II.6.4. Code Generation - Imperative Style - Application of DO

- **Goal:** produce the follwing code:

```
cout << "2^1 =  2"  << endl;
cout << "2^2 =  4"  << endl;
cout << "2^3 =  8"  << endl;
cout << "2^4 = 16"  << endl;
```

- Statement-class:

```
template <int i=1>
struct printPower2 {
  enum { m=i }; // export i
  StaticInt<i> I;
  static void exec() {
    cout<< "2^"<<i<<"="<<\
           power(2,I)<<endl;
  }
  typedef printPower<i+1> Next;
};
```

- Condition-class

```
template <int n>
struct powerCond {
  template <typename Statement>
  struct Code {
    enum { RET = Statement::m < n }
  };
};
```

Solution: `DO<printPower2<>,powerCond<4> >::exec()`

II.6.4. Code Generation - Imperative Style - Implementation of WHILE

```
struct Stop {
  static void exec() {};
};

template <class Condition, class Statement>
struct WHILE {
  static void exec() {
    IF<Condition::template Code<Statement>::RET,
        Statement,
        Stop
       >::RET::exec();
     typedef typename Statement::Next NextStatement;
     IF   <Condition::template Code<Statement>::RET,
           WHILE<Condition, NextStatement >,
           Stop
          >::RET::exec();
   }
};
```

II.6.4. Code Generation - Imperative Style - FOR

```
struct Statement {
  template <int land>
  struct Code {
    static void exec() {
      LOKALISIERE_BANKAUTOMAT<(Land)land>::RET bankautomat;
      teste(bankautomat);
    }
  };
};

// ...
FOR<Deutschland,LessEqual,USA,1,Statement>::exec();
```

II.6.4. Code Generation - Implementation of `Less` and `LessEqual`

```
struct Less {
  template<int x, int y>
  struct Code {
    enum { RET = x<y };
  };
};

struct LessEqual {
  template<int x, int y>
  struct Code {
    enum { RET = x<=y };
  };
};
```

II.6.4. Code Generation - Imperative Style - Implementation of `FOR`

```
template <int from, class Compare, int to, int by, class Statement>
struct FOR {
   static void exec() {
     typedef typename Statement::Code<from> Code_;
     IF<Compare::template Code<from,to>::RET,
        Code_,
        Stop
       >::RET::exec();
     IF<Compare::template Code<from,to>::RET,
        FOR<from+by,Compare,to,by,Statement>,
        Stop
      >::RET::exec();
   }
};
```

II.6.4. Code Generation - Nesting Loops

```
int m[3][4] =  {  { 1, 2, 3, 4},{ 5, 6, 7, 8},{ 9,10,11,12} };

struct OuterLoop {
  template <int i>
  struct Code {
    struct InnerLoop {
      template <int j>
      struct Code {
        static void exec() {
          cout << i << ',' << j << ": " << m[i][j];
          if (j == 3)
            cout << endl;
          else cout << '\t';
        }
      };
    };
    static void exec() {
      FOR<0,LessEqual,3,+1,InnereSchleife>::exec();
    }
  };
};
// …
FOR<0,LessEqual,2,+1,AeussereSchleife>::exec();
```

II.6.5. Code Generation - Summary

- Regarding memory usage and compile time, sophisticated code generating TMFs are most efficient
- `DO`, `WHILE` and `FOR` are very descriptive (higher intentionality)
- `DO`, `WHILE` and `FOR` are similar to their runtime counterparts and allow for an easy adaptation of existing algorithms
- `DO`, `WHILE` and `FOR` are recursively implemented. However, their use doesn’t reveal this implementation detail

Note 21: *Turing Completeness of TMFs*

- A ***Turing Complete*** language is a language with at least a conditional and a while-loop construct - such a language can be used to implement a Turing machine.
- Under Church's conjecture, any language in which a Turing machine can be simulated is ***powerful enough to perform any realizable algorithm***

The static C++ level is Turing complete, there are no theoretical limits to what you can implement at that level

Overview

- What is template metaprogramming?
- Metainformation
- Computing values
- Computing types
- Functional Flavor of the Static Level
- Code generation
- *Outlook and evaluation*
- References

II.7.1. Outlook and Evaluation - Further Topics

Further Topics

- Expression-Templates
  - generation of parse tress for expressions like
    `e = m1 + m2 +m3` // for Matrices
  - optimizing code through transformation of parse trees
  - implementation of domain-specific languages (e.g. **FACT!**)
- Implementation of functional language - execution during compile-time (e.g. LISP kernel)
- Generation of meaningful error messages during compile time (concept checking)
- Partial Evaluation

II.7.1. Outlook and Evaluation - Further Topics

Further Topics

- Configuration generators
- Lazy and full evaluation of TMF expression templates
- Multiparadigm programming with C++
- Template Metaprogramming allows for fascinating applications like
  - generative programming in C++
  - scientific computing

II.7.2. Outlook and Evaluation - History

- Randomly discovered - there was no intend by the language designers
- First template meta program was written by ***Erwin Unruh***:

```
template <int i> struct D { D (void*); operator int(); };
template <int p,int i> struct is_prime {
  enum { prim= (p%i) && is_prime<(i>2 ? p :0),i-1>::prim }
};

template <int i> struct Prime_print {
  Prime_print<i-1> a;
  enum { prim = is_prime<i,i-1>::prim }
  void f() { D<i> d = prim; }
};

struct is_prime<0,0> { enum {prim=1}; };
struct is_primt<0,1> { enum {prim=1}; };
struct Prime_print<2> { enum {prim=1}; void f() { D<2> d = prim; } };
#ifndef LAST
#define LAST 10
#endif
main {
  Prime_print<LAST> a;
}
```

II.7.3. Outlook and Evaluation - Problems

- Several problems exist
  - debugging
  - error messages
  - readability of code
  - compile time
  - compiler limitations

II.7.3. Outlook and Evaluation - Debugger

- There is no debugger for TMF's
- Workarounds for producing meaningful error messages do not ever work optimal
- **`typename`** expressions can become very long
- Compiler cuts long typenames
- Meta programs result in very long typenames

II.7.3. Outlook and Evaluation - Example for a very long typename

Typename for the matrix expression (A+B)*C

```
MultiplicationExpression<class LazyBinaryExpression<class AdditionExpression<class
MatrixICCL::Matrix<class MatrixICCL::BoundsChecker<class MatrixICCL::ArrFormat<class
MatrixICCL::StatExt<struct MatrixDSL::int_number<int,7>,struct
MatrixDSL::int_number<int,7>>,class MatrixICCL::Rect<class MatrixICCL::StatExt<struct
MatrixDSL::int_number<int,7>,struct MatrixDSL::int_number<int,7>>>,class
MatrixICCL::Dyn2DCContainer<class MATRIX_ASSEMBLE_COMPONENTS<class
MATRIX_DSL_ASSIGN_DEFAULTS<class MATRIX_DSL_PARSER<struct MatrixDSL::matrix<int,struct
MatrixDSL::structure<struct MatrixDSL::rect<struct MatrixDSL::stat_val<struct
MatrixDSL::int_number<int,7>>,struct MatrixDSL::stat_val<struct
MatrixDSL::int_number<int,7>>,struct MatrixDSL::unspecified_DSL_feature>,struct
MatrixDSL::dense<struct MatrixDSL::unspecified_DSL_feature>,struct MatrixDSL::dyn<struct
MatrixDSL::unspecified_DSL_feature>>,struct MatrixDSL::speed<struct
MatrixDSL::unspecified_DSL_feature>,struct MatrixDSL::unspecified_DSL_feature,struct
MatrixDSL::unspecified_DSL_feature,struct MatrixDSL::unspecified_DSL_feature,struct
MatrixDSL::unspecified_DSL_feature>>::DSLConfig>::DSLConfig>>>>,class
MatrixICCL::Matrix<class MatrixICCL::BoundsChecker<class MatrixICCL::ArrFormat<class
MatrixICCL::StatExt<struct MatrixDSL::int_number<int,7>,struct
MatrixDSL::int_number<int,7>>,class MatrixICCL::Rect<class MatrixICCL::StatExt<struct
MatrixDSL::int_number<int,7>,struct MatrixDSL::int_number<int,7>>>,class
MatrixICCL::Dyn2DCContainer<class MATRIX_ASSEMBLE_COMPONENTS<class
MATRIX_DSL_ASSIGN_DEFAULTS<class MATRIX_DSL_PARSER<struct MatrixDSL::matrix<int,struct
MatrixDSL::structure<struct MatrixDSL::rect<struct MatrixDSL::stat_val<struct
MatrixDSL::int_number<int,7>>,struct MatrixDSL::stat_val<struct
MatrixDSL::int_number<int,7>>,struct MatrixDSL::unspecified_DSL_featu      uct
MatrixDSL::dense<struct MatrixDSL::unspecified_DSL_feature>,struct Ma  . . .
```

to be continued ...

II.7.3. Outlook and Evaluation - Readability

- Reading and understanding TMFs needs some practice
- TMF like EVAL_TABLE increase readability but extend compile time
- A lot of “**<**“ and “**>**” increase possibility for errors.

II.7.3. Outlook and Evaluation - Compile time

- C++ compilers are not optimized for template metaprogramming
- compile times increase significantly (sometimes dramatically)
- template meta programs get interpreted, not compiled
- compile time depends on complexity of meta program, programming style and the implementation of the compiler

II.7.3. Outlook and Evaluation - Compiler Limits

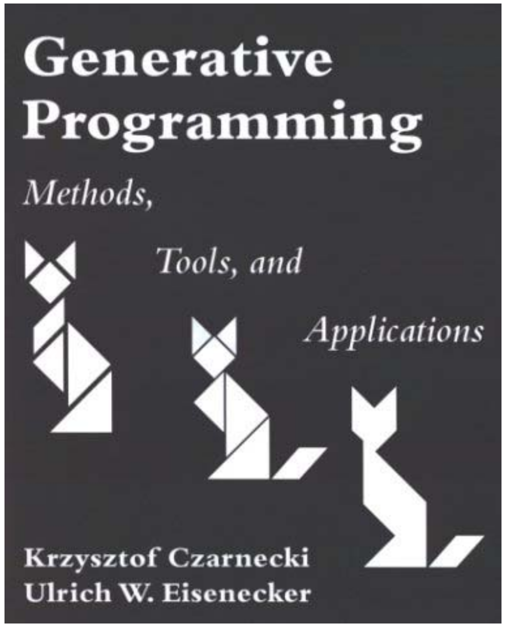
- limitations vary among different compilers
- Execution of template meta programs are a byproduct of type generation and type inference
- complex computations result in complex types
- The C++ standard requires only 17 nested template instantiations (Annex AA)

Overview

- What is template metaprogramming?
- Metainformation
- Computing values
- Computing types
- Functional Flavor of the Static Level
- Code generation
- Outlook and evaluation
- *References*

II.8. References

- *Czarnecki, Eisenecker*: **Generative Programming: Methods, Techniques, and Applications**, Addison-Wesley, 2000



- *Eisenecker, Czarnecki*: **Template-Metaprogrammierung - eine Einführung**, OBJEKTspektrum 3/99

- *Knaupp, Eisenecker, Czarnecki*: **Mit Tabellen zur Entscheidung**, OBJEKTspektrum 5/99

III. Advanced Techniques

λ☺

Jörg Striegnitz
Research Centre Jülich GmbH
j.striegnitz@fz-juelich.de

Overview

- *Template Metaprogramming & Data Structures*
- Type Checking
- Guiding Code Production with Data Structures
- Expression Templates and PETE
- Advanced Metaprograms
- References

III.1.1. TMP & Data Structures - Overview

- Simple Variables
- Singly Linked Lists
- Trees

III.1.1. TMP & Data Structures - Simple Variables

- varaibles are **`typedef`** names and integral constants

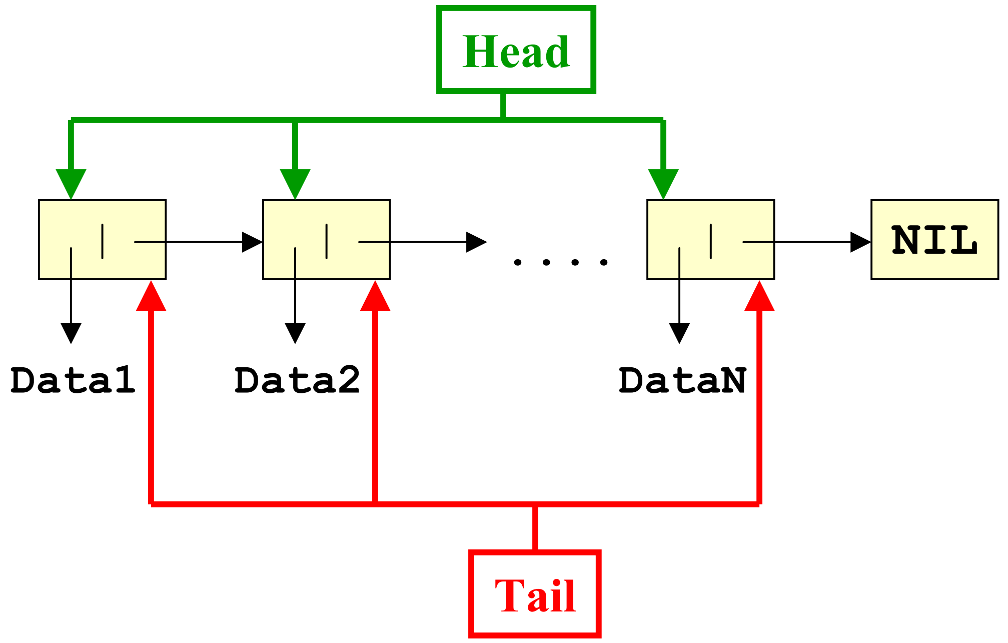
```
struct StopTag {};

typedef PrintPower2<5> DefaultStatement;

const int UNROLL_DEPTH = 2;
```

- single assignment rule:

Variables are initialized ones, their value cannot be changed

III.1.2. TMP & Data Structures - Singly Linked Lists

- Use **nested templates** to represent list.
- View list as chain of head-and-tail pairs:



- Head contains list-element, Tail contains rest of list.
- Tail of last element is **`NIL`** - the empty list

III.1.2. TMP & Data Structures - Singly Linked Lists (SLL)

- Remember the ***functional flavor*** !
- With LISP, lists get created through the `cons` contructor:

```
(cons 0 (cons 0 (cons 7 (cons 42 nil))))
```

creates the list `( 0 0 7 42 )`

- Make **`cons`** a template struct and **`nil`** a struct:

```
CONS<0, CONS<0, CONS<7, CONS<42, NIL> > > >
```

```
struct NIL { enum {head=(~(~0u) >> 1)} };

template <int headVal,typename TAIL>
struct CONS {
   enum { head=headVal };
   typedef TAIL Tail_t;
};
```

III.1.2. TMP & Data Structures - SLL - Length of List

```
# length: [a] -> int
length []     = 0
length (h:t)  = 1 + length t
```

Argument type becomes template parameter

```
template <typename LIST> struct Length;

template <>
struct Length<NIL> {
  enum { RET = 0 };
};


template <typename LIST>
struct Length {
  enum { RET = 1 + Length<typename LIST::Tail_T>::RET }
};
```

III.1.2. TMP & Data Structures - SLL - Length of List

```
# length: [a] -> int
length []     = 0
length (h:t) = 1 + length t
```

Integral result-type becomes **enum** member

```
template <typename LIST> struct Length;

template <>
struct Length<NIL> {
  enum { RET = 0 };
};


template <typename LIST>
struct Length {
  enum { RET = 1 + Length<typename LIST::Tail_T>::RET }
};
```

III.1.2. TMP & Data Structures - SLL - Length of List

```
# length: [a] -> int
length []     = 0
length (h:t)  = 1 + length t
```

Every pattern becomes a template / template specialization

```
template <typename LIST> struct Length;

template <>
struct Length<NIL> {
  enum { RET = 0 };
};


template <typename LIST>
struct Length { // struct Length<LIST>
  enum { RET = 1 + Length<typename LIST::Tail_T>::RET }
};
```

III.1.2. TMP & Data Structures - SLL - Length of List

```
# length: [a] -> int
length []     = 0
length (h:t)  = 1 + length t
```

Statement becomes assignment to **RET** / **typedef** definition for **RET**

```
template <typename LIST> struct Length;

template <>
struct Length<NIL> {
   enum { RET = 0 };
};


template <typename LIST>
struct Length {
   enum { RET = 1 + Length<typename LIST::Tail_T>::RET }
};
```

III.1.2. TMP & Data Structures - SLL - Checking for a List Member

```
# member: [a] -> a -> bool
member []     a = False
member (h:t) a = h==a || member t a
```

```
template <typename LIST,int a> struct Member;

template <int a>
struct Member<NIL,a> {
  enum { RET = false };
};


template <typename LIST,int a>
struct Member {
  enum { RET = (a == LIST::head) ||
                Member<typename LIST::Tail_T,a>::RET };
};
```

III.1.2. TMP & Data Structures - SLL - Append

```
# append: [a] -> [a] -> [a]
append [] a     = a
append (h:t) a = h:append t a
```

```
template <typename LIST1,typename LIST2> struct append;

template <typename LIST2>
struct append<NIL,LIST2> {
  typedef LIST2 RET;
};


template <typename LIST1,typename LIST2>
struct append {
  typedef CONS<LIST1::head,
               typename append<typename LIST1::Tail_T, LIST2>::RET
              > RET;
};
```

III.1.2. TMP & Data Structures - SLL - Storing More Complex Meta-Data

- Producing lists of elements of arbitrary types:

```
struct NIL { };

template <typename HEAD,typename TAIL>
struct CONS {
   typedef HEAD Head_T;
   typedef TAIL Tail_T;
};
```

- `length` and `append` still work !
- Slightly different version of `member` needed now !

III.1.2. TMP & Data Structures - SLL - Member II

```
# equal a -> b -> bool
equal x y = False
equal x x = true
```

```
template <typename T1,typename T2>
struct EQUAL          { enum { RET = false}; };
template <typename T>
struct EQUAL <T,T> { enum { RET = true }; };
```

```
# member: [a] -> a -> bool
member []     a = False
member (h:t) a = h==a || member t a
```

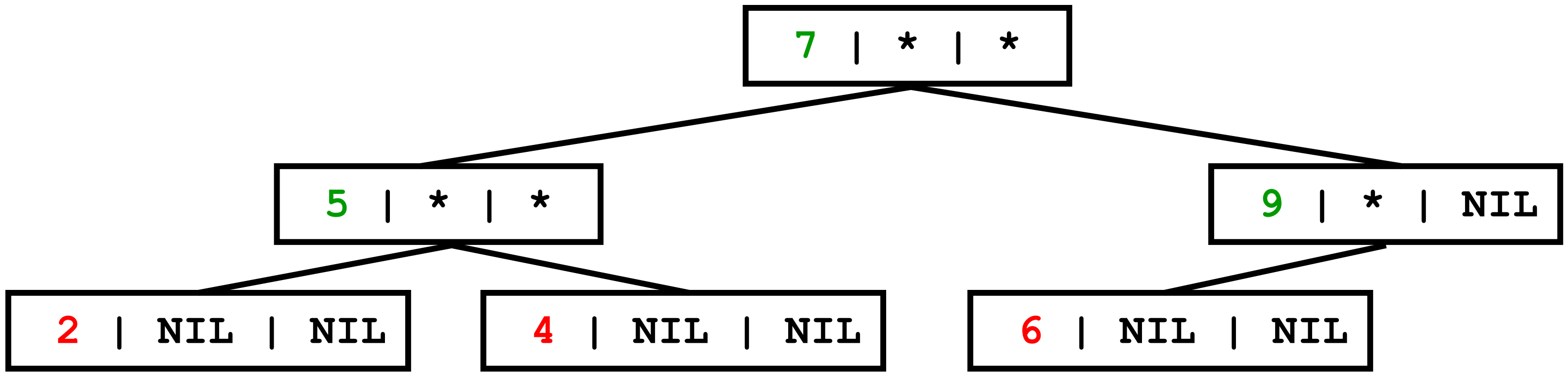
```
template <typename LIST,typename M> struct Member;

template <typename M>
struct Member<NIL,M> {
  enum { RET = false };
};

template <typename LIST,typename M>
struct Member {
  enum { RET = EQUAL<typename LIST::Head_T,M>::RET ||
               Member<typename LIST::Tail_T,M>::RET };
};
```

III.1.3. TMP & Data Structures - Trees

- ***Idea:*** flatten tree into a list
- First, define tree and nodes:

```
struct NIL {};

template <int value,typename LEFT,typename RIGHT>
struct BinaryNode {
  typedef LEFT  Left_T;
  enum { Value=value };
  typedef RIGHT Right_T;

}
```

III.1.3. TMP & Data Structures - Trees - A Binary Tree



```
BinaryNode<7,
           BinaryNode<5,
                      BinaryNode<2,NIL,NIL>,
                      BinaryNode<4,NIL,NIL>
                     >,
           BinaryNode<9,
                      BinaryNode<6,NIL,NIL>,
                      NIL
                     >
          >
```

III.1.3. TMP & Data Structures - Trees - Depth of a Tree

```
# depth: Tree * -> int
depth NIL                   = 0
depth BinaryNode l a r = 1 + max depth(l) depth(r)
```

```
template <int v1,int v2>
struct Max {
  enum { RET = (v1 > v2) ? v1 : v2 };
};

template <typename T> struct Depth;

template <> struct Depth<NIL> {
   enum { RET = 0 };
};

template <typename T> struct Depth {
   enum { RET = 1 + Max<Depth<typename T::Left_T>::RET,
                        Depth<typename T::Right_T>::RET
                       >::RET;
        };
};
```

III.1.3. TMP & Data Structures

- Nested templates can be used to represent data
- Lists, Trees, Nodes, Graphs, etc. possible
- TMFs act on data
- tight relation to functional programming
- type names may get even longer
- increased compile time

Overview

- Template Metaprogramming & Data Structures
- *Type Checking*
- Guiding Code Production with Data Structures
- Expression Templates and PETE
- Advanced Metaprograms
- References

III.2. Type Checking - Member

• ***Problem:*** It is not obvious that **`member`** is acting on a list !

```
template <typename LIST,typename M> struct Member;

template <typename M>
struct Member<NIL,M> {
   enum { RET = false };
};


template <typename LIST,typename M>
struct Member {
   enum { RET = EQUAL< typename LIST::Head_T,M>::RET ||
                Member<typename LIST::Tail_T,M>::RET };
};
```

name of parameter as a hint

• Calling **`Member`** with other *types* results in complicated error messages

III.2. Type Checking - Error

```
//..
const bool inList = Member<double,double>::RET; // line 133
```

• `g++ 2.95.2` generates the following error message:

```
List.cc: In instantiation of ‘Member<double,double>’
list.cc:133: instantiated from here
list.cc:133: ‘double’ is not a class, struct, or union type
list.cc:133: template argument 1 is invalid
list.cc:133: ‘double’ is not a class, struct, or union type
list.cc:133: template argument 1 is invalid
list.cc:133: enumerator value of ‘RET’ not integer constant
```

☹ Oops: the real problem is that **`double`** is not a **`CONS`** type - g++ doesn’t tell ...

• Understanding error message depends on accessibility of **`Member`**
• But: no pointer to definition of **`Member`**

III.2. Type Checking - Generating Errors / Error Templates

• Declare a template structure *whose name is the error message to generate*

```
template <typename T>
struct ERROR__EXPECTED_A_LIST_TYPE;
```

• **In case of a type-error**: create an instance of the *error template*

```
template <typename LIST,typename M>
struct Member2 {
 ERROR__EXPECTED_A_LIST_TYPE<LIST> error;
};
```

Instantiating this template will cause an error because **ERROR__EXPECTED_A_LIST_TYPE** is **undefined** !

• template specializations for *type-safe* cases:

```
template <typename M>
struct Member2<NIL, M> { ... }:

template <typename M,typename HEAD,typename TAIL>
struct Member2<CONS<HEAD,TAIL>, M> { ... };
```

III.2. Type Checking - Generating Errors

```
List2.cc: In instantiation of `Member2<double,double>':
List2.cc:294:   instantiated from here
List2.cc:119: invalid use of undefined type
              `struct ERROR__EXPECTED_LIST_TYPE<double>'
List2.cc:116: forward declaration of
              `struct ERROR__EXPECTED_LIST_TYPE<double>'
List2.cc:121: confused by earlier errors, bailing out
```

This error message contains a lot more information:

- The real **error message** appears and contains a hint to the **wrong argument**
- Line **294** contains the **erroneous call** to the `Member2` TMF
- The **definition of `Member`** starts at line **119**

☹ Still not an optimal solution. Unfortunately, the best one I have at hand ....

III.2. Type Checking - Generating Error Messages: A more complicated example I

- Another possible solution:

```
template <typename M>
class Error__First_parameter_must_be_a_LIST {
  // IGNORE_THIS is private in Member2 !
  typedef typename M::IGNORE_THIS T;
  enum { RET = T::IGNORE_THIS_AS_WELL };
};
```

```
template <typename NON_LIST,typename M>
struct Member3 {
private:
  struct IGNORE_THIS { enum { IGNORE_THIS_AS_WELL }; };
public:
  enum { RET = Error__First_parameter_must_be_a_LIST<Member3>::RET
         };
};
```

III.2. Type Checking - Generating Error Messages: A more complicated example II

```
//..
const bool inList = Member3<double,double>::RET; // line 165
```

- `g++ 2.95.2` generates the following error message:

```
List.cc: In instantiation of
         Error__First_parameter_must_be_CONS<
            Member3<double,double>
         >
list.cc:165: instantiated from Member3<double,double>
list.cc:165: instantiated from here
list.cc:165: no type named ‘IGNORE_THIS’ in struct
             ‘Member3<double,double>’
list.cc:165: enumerator ‘RET’ not integer constant
list.cc:52:  no type named ‘IGNORE_THIS’ in struct
             ‘Member3<double,double>’
```

- Message starts with “real” error
- pointer to implementation of **`Member3`**
- still difficult to read

Overview

- Template Metaprogramming & Data Structures
- Type Checking
- *Guiding Code Production with Data Structures*
- Expression Templates and PETE
- Advanced Metaprograms
- References

III.3. Guiding Code Production - SLL - Polymorphic Lists (Runtime)

• Developing lists that store values of arbitrary types:

```
struct NIL { };

template <typename HEAD,typename TAIL>
struct CONS {
   typedef HEAD Head_T;
   typedef TAIL Tail_T;

   CONS(const HEAD_& h,const TAIL& t) : head(h),tail(t) {}
   CONS(const CONS& rhs) : head(rhs.head),tail(rhs.tail) {}
   CONS& operator=(const CONS& rhs);

   const Head_T& getHead() const { return head; }
   const Tail_T& getTail() const { return tail; }

private:
   HEAD head;
   TAIL tail;
};
```

Manipulate runtime data - runtime behaviour

Static information

Store instances of parameters - runtime information !

III.3. Guiding Code Production - SLL - Polymorphic Lists (Runtime)

• convenience function to build lists

```
template <typename HEAD,typename TAIL>
inline CONS<HEAD,TAIL> cons(const HEAD& h,const TAIL& t) {
  return CONS<HEAD,TAIL>(h,t);
}
```

Example:

```
complex<double> c(1,2), d(3,4);
std::string s(“Hallo”);
std::vector v;

cons(c, cons(d, cons(s, cons(v, NIL() )))); // Temporary !
```

```
CONS< complex<double>,
      CONS< complex<double>,
            CONS< std::string,
                  CONS< std::vector, NIL >
                >
          >
    >
```

III.3. Guiding Code Production - Generating a List

- use function to produce temporary
- overload function to support lists of different size

```
template <typename A1,typename A2,typename A3>
CONS<A1,CONS<A2,CONS<A3,NIL> > >
mkList(const A1& a1,const A2& a2,const A3& a3) {
  return cons(a1,cons(a2,cons(a3,NIL() )));
}

template <typename A1,typename A2,typename A3,typename A4>
CONS<A1,CONS<A2,CONS<A3,CONS<A4,NIL> > > >
mkList(const A1& a1,const A2& a2,const A3& a3,const A4& a4) {
  return cons(a1,cons(a2,cons(a3,cons(a4,NIL() ))));
}
```

III.3. Guiding Code Production - Traversing a List

```
# foreach: [a] -> (a->b) -> [b]
foreach [] f    = []
foreach (h:t) f = (f h):foreach t f
```

• applying **`f`** to the empty list **`[]`** yields the empty list **`[]`**

• If the list is separable into a head **`h`** and a tail **`t`** we return a list whose
  – first element results from applying **`f`** to **`h`**
  – the tail of the list results from applying **`foreach`** to it

• passing anything else but a list to **`Foreach`** should yield a type error

```
template <typename LIST,typename fTAG=IDTag>
struct Foreach {
 ERROR__EXPECTED_A_LIST_TYPE<LIST> error;
};
```

III.3. Guiding Code Production - Traversing a List

```
# foreach: [a] -> (a->b) -> [b]
foreach [] f     = []
foreach (h:t) f = (f h):foreach t f
```

• Return the empty list **`NIL`**, if **`foreach`** is applied to **`NIL`**

Static part - compute result-type

```
template <typename F>
struct Foreach<NIL,F> {

  typedef NIL RET;

  static inline
  RET apply(const NIL& n) {
    return n;
  }
};
```

Runtime part - compute value

III.3. Guiding Code Production - Traversing a List

```
# foreach: [a] -> (a->b) -> [b]
foreach [] f     = []
foreach (h:t) f = (f h):foreach t f
```

Static part - compute result-type

```
template <typename HEAD,typename TAIL,typename fTAG>
struct Foreach<CONS<HEAD,TAIL>, fTAG> {

  typedef CONS< typename ApplyFunctor<HEAD,fTAG>::RET,
                typename Foreach<TAIL, fTAG>::RET
              > RET;

  static inline
  RET apply(const CONS<HEAD,TAIL>& l,const fTAG& f) {
     return cons(
            ApplyFunctor<HEAD,fTAG>::apply(l.getHead(),f),
            Foreach<TAIL,F>::apply(l.getTail(),f)
          );
   }
};
```

Runtime part - compute value

III.3. Guiding Code Production - Traversing a List

- Indirectly calling the functor has some advantages

```
template <typename T,typename fTAG>
struct ApplyFunctor {};
```

- Per default, we apply the identity function:

```
struct IDTag {};

template <typename T>
struct ApplyFunctor<T,IDTag> {
   typedef T RET;
   static inline
   RET apply(const T& t,const IDTag&) { return t; }
};
```

- As for **`cons`**, we introduce a convenience function:

```
template <typename LIST,typename fTAG>
inline typename Foreach<LIST,F>::RET forEach(const LIST& l,
                                             const fTAG& f){
   return Foreach<LIST,fTAG>::apply(l,f);
}
```

III.3. Guiding Code Production - Traversing a List

• *Example [Inserting the values of a list into a stream]*:

```
struct PrintValues {
  PrintValues(ostream& _o) : o(_o) {}
  ostream& getStream() const { return o; }
  std::ostream& o;
};

template <typename T>
struct ApplyFunctor<T,PrintValues > {
  typedef T RET;
  static inline RET apply(const T& n,const PrintValues& p) {
      p.getStream() << "[" << n << "]";
      return n;
    }
  };
};
…
std::complex<double> c(1.0,3.0);
std::string          s="Hello";
double               d=2.0;
forEach( mkList(c,d,s), PrintValues(std::cout) );
```

III.3. Guiding Code Production - Traversing a List

```
std::complex<double> c(1.0,3.0);
std::string          s=“Hello”;
double               d=2.0;
forEach( mkList(c,d,s), PrintValues(cout) );
```

$\downarrow$ Transformed during compile time

```
std::complex<double> c(1.0,3.0);
std::string s=“Hello”;
double d=2.0;
cout << “[” << s << “]”;
cout << “[” << d << “]”;
cout << “[” << c << “]”;
```

- Why reverse order (List is **`(c d s)`**) ?
- C++ does eager evaluation (call by name)

III.3. Guiding Code Production - Mixing Compile Time and Runtime

- ***Task:*** find a value of type **`T`** in a polymorphic list
- This only makes sense, if the list contains any value of type **`T`**

```
template <typename LIST,typename A>
bool isMember( const LIST& l, const A& m ) {
  const bool TYPE_IN_LIST = Member2<LIST,A>::RET;
  if (! TYPE_IN_LIST) {
    return false;
  }
  else {
  // ... search the item
  };
};
```

- Use code selection to move execution of **`if`** into compile time

III.3. Guiding Code Production - Mixing Compile Time and Runtime

```
template <typename LIST,typename A>
struct returnFALSE {
  static inline bool exec(const LIST&,const A&) {
    return false;
  }
};

template <typename LIST,typename A>
struct searchItem {
  static inline bool exec(const LIST& l,const A& m) {
  // .. search the item
  };
};

template <typename LIST,typename A>
bool isMember( const LIST& l, const A& m ) {
  enum { TYPE_IN_LIST = Member2<LIST,A>::RET };
  return IF< TYPE_IN_LIST,  searchItem<LIST,A>,
                           returnFALSE<LIST,A>
           >::RET::exec(l,m);
};
```

III.3. Guiding Code Production - Mixing Compile Time and Runtime

- Only need to compare items of same type !
- Build new list that only contains values of desired type

```
# filter: (a -> bool) -> [a] -> [a]
filter p []     = []
filter p (h:t) = h:filter p t, if p h
               = filter p t,   otherwise
```

```
template <typename HEAD,typename TAIL,typename P>
struct Filter<P, CONS<HEAD,TAIL> > {
  typedef typename
  IF< P::template apply<HEAD>::RET,
      CONS<HEAD, Filter<P,TAIL> >,
      Filter<P,TAIL>
    >::RET RET;
  // ... Runtime part
};
```

III.3. Guiding Code Production - Mixing Compile Time and Runtime

```
template <typename P,typename HEAD,typename TAIL>
struct Filter<P, CONS<HEAD,TAIL> > {
//...
  struct return_H_F { // include HEAD
    static inline
    RET apply(const CONS<HEAD,TAIL>& l) {
      return cons(l.getHead(),
                  Filter<P,TAIL>::apply( l.getTail() )
                 );
    }
  };
  struct return_F {  // remove HEAD
    static inline
    RET apply(const CONS<HEAD,TAIL>& l) {
      return Filter<P,TAIL>::apply( l.getTail() );
    }
  };
  //..
};
```

III.3. Guiding Code Production - Mixing Compile Time and Runtime

```
template <typename P,typename HEAD,typenam TAIL>
struct Filter<P, CONS<HEAD,TAIL> > {
//..
  static inline
  RET apply(const CONS<HEAD,TAIL>& l) {
    return IF<P::template apply<HEAD>::RET,
              return_H_F,
              return_F
             >::apply(l);
  }
//..
};

// ... handle case for empty list ...
```

```
template <typename P,typename LIST>
typename Filter<P,LIST>::RET filter(const P& p, const LIST& l) {
  return Filter<P,LIST>::apply(l);
};
```

III.3. Guiding Code Production - Mixing Compile Time and Runtime

- ***Example:*** filtering all complex numbers:

```
struct isComplex {
  template <typename T> struct apply {
    enum { RET = false };
  };
  template <typename T> struct apply<complex<T> > {
    enum { RET = true };
  };
};
complex<double> c(1,2),d(3,4);
double x,y,z;

forEach( filter(isComplex(),
                mkList(c, d, x, y, z)
               ),
          printTag()
        );
```

Transformed during compile time

```
cout << "[" << d << "]";
cout << "[" << c << "]";
```

III.3. Guiding Code Production - Finding Element in List (Runtime)

```
# member:  [a] -> a -> bool
member []     a = false
member (h:t)  a = h==a || member t a
```

```
template <typename LIST,typename M>
struct Member {
   enum { RET = EQUAL<typename LIST::head,M>::RET ||
                Member<typename LIST::tail,M>::RET };
   static inline
   bool apply(const LIST&l, const M& m) {
      return LIST::getHead(l) == m ||
             Member<typename LIST::Tail_T,M>::apply(
             LIST::getTail(l), m);

   };
};
```

constraint: **operator==** needs to be defined !

- **operator==** maybe undefined, especially if **M != Head_T**
- **->** return **false** whenever **M != Head_T**

III.3. Guiding Code Production - Finding Element in List (Runtime)

```
template <typename HEAD,typename TAIL,typename M>
struct Member<CONS<HEAD,TAIL>, M> {
  //..
  struct returnFALSE {
    static inline bool exec(const CONS<HEAD,TAIL>& l,const M& m) {
      return false;
    }
  };
  struct returnCOMP {
    static inline bool exec(const CONS<HEAD,TAIL>& l, const M& m) {
      return l.getHead() == m;
    }
  };
  static inline
  bool apply(const CONS<HEAD,TAIL>& l, const M& m) {
    return IF< EQUAL<HEAD,M>::RET,
               returnCOMP, returnFALSE >::RET::exec(l,m) ||
           Member<TAIL,M>::apply(l.getTail(), m);
  }
};
```

III.3. Guiding Code Production - Finding Element in List (Runtime)

```
complex<double> a(2,3);
string b="Hello";
double c=2.0;
double d=3.0;
complex<double> e(4,5);
member( filter(isComplex(),
               mkList(a,b,c,d)
              ),
         e
       ) ? cout << "Yes" : cout << "No";
```

Transformed
during compile time

$\longrightarrow$

```
a==e ? cout << "Yes" : cout << "No";
```

☺ **Cool optimization !**

☹ **Long compile time !**

III.3. Guiding Code Production - Summary

- TMP provides good means to optimize programs
- TMP to do partial evaluation
- Expression templates use GCP
- Implementing sub language (**FACT!**)
- compile times increase
- meta programs + data structures result in very very long typenames

Overview

- Template Metaprogramming & Data Structures
- Type Checking
- Guiding Code Production with Data Structures
- *Expression Templates and PETE*
- Advanced Metaprograms
- References

III.4. Expression Templates - The Problem I

As shown on section **I**, operator overloading introduces a lot of temporaries and copy constructor calls:

```
complex<double> a,b,c,d,result;
result = a * b - (c + d);
```

How many temporaries are created here ?

- a temporary to hold the value of **`a * b`**
- a temporary to hold the result of **`c + d`**
- a temporary to hold the result of subtraction

- **may be very expensive !**

```
// C - code:
complex<double> tmp1;
operator*(tmp1,a,b);

complex<double> tmp2;
operator+(tmp2,c,d);

complex<double> tmp3;
operator-(tmp3,tmp1,tmp2);

result.complex<double>::operator=(tmp3);

tmp3.complex<double>::~complex<double>;
tmp2.complex<double>::~complex<double>;
tmp1.complex<double>::~complex<double>;
```

III.4. Expression Templates - The Problem II

```
template <int size>
struct UArray {
  // ... Constructors / destructor / other stuff
private:
  double data[size];
};

template <int size>
UArray<size> operator+(const UArray<size>& a,const UArray<size>& b) {
  array tmp;
  for (int i=0 ; i<size; i++) {
    tmp[i]= a[i] + b[i];
  }
  return tmp;
}
//... almost the same for opertaor-, operator*, operator/, ...
```

```
UArray<10> a,b,c,d,e;
d = (a+b) / (c*d);
```

Four loops (one for the assignment) !

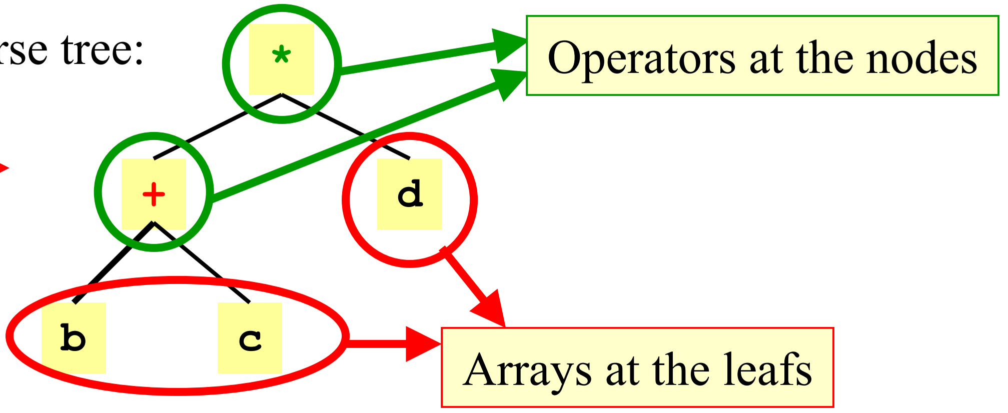
Desirable : **single loop** !

For brevity we omit the integer template parameter for the rest of this section

III.4. Expression Templates - The Fundamental Idea

• For every expression; construct its parse tree:

```
UArray a,b,c,d;
a = ((b + c)  * d);
```



• Traverse tree **size** times and perform actions according to tree nodes

```
template <typename ParseTree>
UArray& UArray::operator=(const ParseTree& t) {
  for (int i=0 ; i< size; i++)
    data[i] = TraverseTreeAndPerformOp( t, i );
}
```

Evaluation of expression is postponed until assignment occurs !

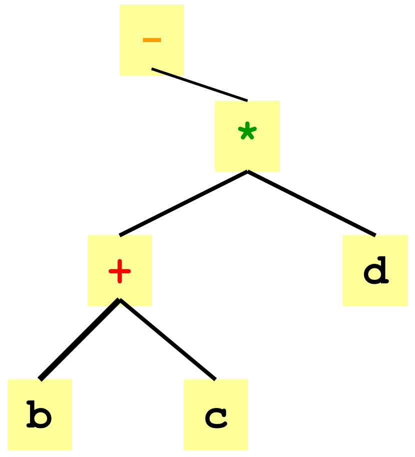
III.4. Expression Templates - The Expression Tree

- The datatypes for the parse tree (expression template tree)

```
Expression = Scalar      a
           | UnaryNode   OpTag Expression
           | BinaryNode  OpTag Expression Expression
           | TernaryNode OpTag Expression Expression Expression
```

- Example:

```
UArray b,c,d;
-((b + c) * d);
```



```
UnaryNode OpUnaryMinus (BinaryNode OpMultiply
                                   (BinaryNode OpAdd UArray UArray)
                                   UArray
                       )
```

III.4. Expression Templates - The Expression Tree

```
Expression = Scalar       a
           | UnaryNode   OpTag Expression
           | BinaryNode  OpTag Expression Expression
           | TernaryNode OpTag Expression Expression Expression
```

```
template <typename T>
struct Scalar;

template <typename OpTag,typename CHILD>
struct UnaryNode;

template <typename OpTag,typename LEFT,typename RIGHT>
struct BinaryNode;

template <typename OpTag,typename LEFT,typename MIDDLE,typename RIGHT>
struct TernaryNode;
```

- Provide constructor, copy constructor
- Store tag and childs
- Provide constant access functions to childs and tag

III.4. Expression Templates - Node Types

- **Example**: Implementation of `UnaryNode`

```
template <typename OP,typename CHILD>
struct UnaryNode {

  UnaryNode(const OP& o,const CHILD& c) : op(o),child(c) {}
  UnaryNode(const UnaryNode& rhs) :
     op(rhs.op),child(rhs.child) {}

  const OP& operation() const { return op; }
  const CHILD& child() const  { return child; }

private:
  OP    op;
  CHILD child;
};
```

III.4. Expression Templates - Tags

```
struct OpUnaryMinus {
  template <typename T>
  inline T operator()(const T& t) const { return -t; }
};
```

```
struct OpMultiply {
  template <typename A1,typename A2>
  inline ??? operator()(const A1& a1,const A2& a2) const {
    return a1 * a2;
  }
};
```

Result-type ? See **section II.4.1**

```
struct OpMultiply {
  template <typename A1,typename A2>
  inline typename BinaryReturn<A1,A2,OpMultiply>::Type_t
      operator()(const A1& a1,const A2& a2) const {
    return a1 * a2;
  }
};
```

• User may specialize **`BinaryReturn`** for his types (if necessary)

III.4. Expression Templates - Computing Return Types

- The default case for **`BinaryReturn`** can reuse the **`Promote`** metafunction that was introduced in **section II4.2**

```
template <typename A1,typename A2,typename OP>
struct BinaryReturn {
   typedef typename Promote<A1,A2>::RET Type_t;
};
```

- For user-defined classes (e.g. matrices) the user could specialize **`BinaryReturn`**:

```
template <typename A1,typename A2>
struct BinaryReturn<Matrix<A1>,Matrix<A2>, OpAdd > {
   typedef Matrix<typename Promote<A1,A2>::RET> Type_t;
};

template <typename A1,typename A2>
struct BinaryReturn<Matrix<A1>,Vector<A2>, OpMultiply > {
   typedef Vector<typename Promote<A1,A2>::RET> Type_t;
};
```

III.4. Expression Templates - Building the Parse Tree I

```
UnaryNode<OpUnaryMinus, UArray>
          operator-(const UArray& a) {
  return UnaryNode<OpUnaryMinus,UArray>(OpUnaryMinus(), a);
};
```

```
// UArray + UArray -> UArray
BinaryNode<OpAdd, UArray, UArray>
          operator+(const UArray& a,const Uarray& b) {
  return BinaryNode<OpAdd,UArray,UArray>(OpAdd(), a, b);
};
```

- **Problem:** we need overloaded versions of `operator+` for **15** other cases (see next slide)

III.4. Expression Templates - Building the Parse Tree II

```
UArray                    + UnaryNode<*,*>
UArray                    + BinaryNode<*,*,*>
UArray                    + TernaryNode<*,*,*,*>

UnaryNode<*,*>            + UArray
BinaryNode<*,*,*>         + UArray
TernaryNode<*,*,*,*>      + Uarray

UnaryNode<*,*>            + UnaryNode<*,*>
UnaryNode<*,*>            + BinaryNode<*,*,*>
UnaryNode<*,*>            + TernaryNode<*,*,*,*>

BinaryNode<*,*,*>         + BinaryNode<*,*,*>
BinaryNode<*,*,*>         + UnaryNode<*,*>
BinaryNode<*,*,*>         + TernaryNode<*,*,*,*>

TernaryNode<*,*,*,*>      + TernaryNode<*,*,*,*>
TernaryNode<*,*,*,*>      + BinaryNode<*,*,*>
TernaryNode<*,*,*,*>      + UnaryNode<*,*>
```

These are all of type `Expression` but the compiler doesn't know - 3 additional cases are sufficient:

```
UArray     + Expression
Expression + UArray
Expression + Expression
```

Use wrapper to indicate type !

III.4. Expression Templates - Building the Parse Tree III

```
template <typename E>
struct Expression {
  typedef E Expression_t;

  Expression(const E& _e) : e(_e) {}
  Expression(const Expression& rhs) : e(rhs.e) {}

  const Expression_t& expression() const { return e; }
private:
  E e;
};
```

• Expression is used to wrap values of type **`UnaryNode`**, **`BinaryNode`**, **`TernaryNode`**

```
Expression< UnaryNode<OpUnaryMinus, UArray> >
Expression< BinaryNode<OpMultiply, UArray, UArray> >
```

III.4. Expression Templates - Building the Parse Tree IV

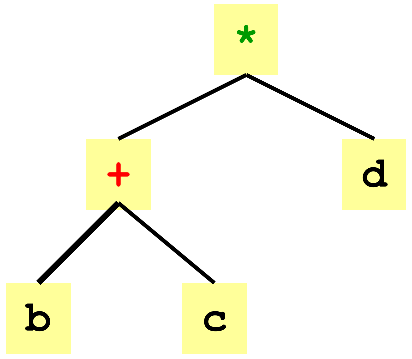
```
Expression< BinaryNode<OpAdd,
                       UArray,
                       UArray> > operator+(const UArray& a,
                                           const UArray& b) {
  return Expression< BinaryNode<OpAdd,UArray,UArray> >(
                     BinaryNode<OpAdd,UArray,UArray>(OpAdd(),a,b)
                     );
};
```

```
template <typename E,int size>
Expression<BinaryNode<OpAdd,
                      E,
                      UArray> > operator+(const Expression<E>& e,
                                          const UArray& b) {
  return Expression< BinaryNode<OpAdd,E,Uarray> >(
                     BinaryNode<OpAdd,E,UArray>(OpAdd(),a,
                                                e.expression())
                     );
};
```

• Notice how the incoming expression gets unwrapped and the result is wrapped ...

III.4. Expression Templates - Building the Parse Tree V

```
UArray<10> a,b,c,d;
(b + c) * d;
```



Resulting Expression Template Tree

```
Expression< BinaryNode< OpMultiply,
                        BinaryNode< OpAdd
                                    UArray,
                                    UArray
                                  >,
                        UArray
                      >
          >
```

Memory (the Object)

```
OpMultiply
  OpAdd
    &b
    &c
    &d
```

III.4. Expression Templates - Building the Parse Tree VI

- What shall we store at the Leafs ? Copies, references, pointers ?
- Allow the developer to specify this (if necessary) !

```
template <typename T>
struct CreateLeaf {
  typedef Scalar<T> Leaf_t;
  static inline Leaf_t make(const T& t) { return Scalar<T>(t); }
};
```

```
template <int size>
Expression< BinaryNode<OpAdd,
                       typename CreateLeaf<UArray >::Leaf_t,
                       typename CreateLeaf<UArray >::Leaf_t
                      >
          > operator+(const UArray& a,const UArray& b) {
  return Expression<...>( BinaryNode<...>(
                             OpAdd(),
                             CreateLeaf<UArray >::make(a),
                             CreateLeaf<UArray >::make(b)
                            );
}
```

III.4. Expression Templates - Building the Parse Tree VII

- Storing copies of an array imposes lots of copy constructors being called
- PETE has special wrapper **`Reference`** to store references at the leaf:

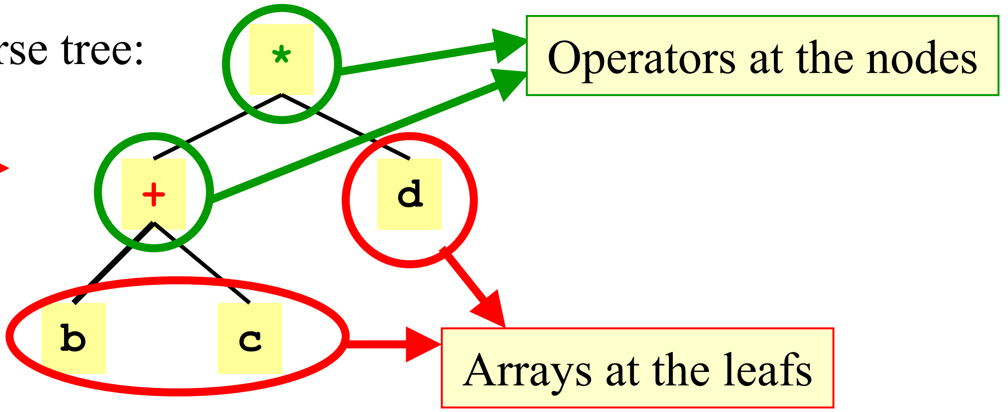
```
template <>
struct CreateLeaf< UArray > {
  typedef Reference< UArray > Leaf_t;
  static inline Leaf_t make(const UArray& a) {
    return Leaf_t( a );
  }
};
```

- PETE offers tool-support for the generation of all the overloaded operators
- To build the expression tree, the user usually only has to specialize from **`CreateLeaf`**

III.4. Expression Templates - The Fundamental Idea - So far we have ...

• For every expression; construct its parse tree:

```
UArray a,b,c,d;
a = ((b + c)  * d);
```



• Traverse tree **size** times and perform actions according to tree nodes

```
template <typename ParseTree>
UArray& UArray::operator=(const ParseTree& t) {
  for (int i=0 ; i< size; i++)
    data[i] = TraverseTreeAndPerformOp( t, i );
}
```

III.4. Expression Templates - The Fundamental Idea - So far we have ...

• For every expression; construct its parse tree:

```
UArray a,b,c,d;
a = ((b + c) * d);
```

Resulting Expression Template Tree

```
Expression< BinaryNode< OpMultiply,
                        BinaryNode< OpAdd
                                    UArray,
                                    UArray
                                  >,
                        UArray
                      >
          >
```

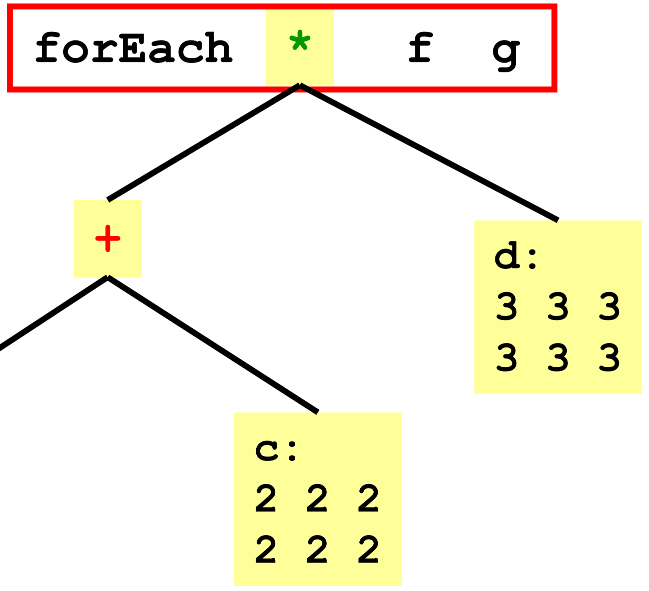
• Traverse tree **`size`** times and perform actions according to tree nodes

```
template <typename E>
UArray& UArray::operator=(const Expression<E>& t) {
  for (int i=0 ; i< size; i++)
    data[i] = TraverseTreeAndPerformOp( t, i );
}
```

`TraverseTreeAndPerformOp` is still missing

III.4. Expression Templates - Traversing the tree

`f`: (applied to LEAFS) return value of array at index **`i`**
`g`: (applied to NODES) perform operation according to node

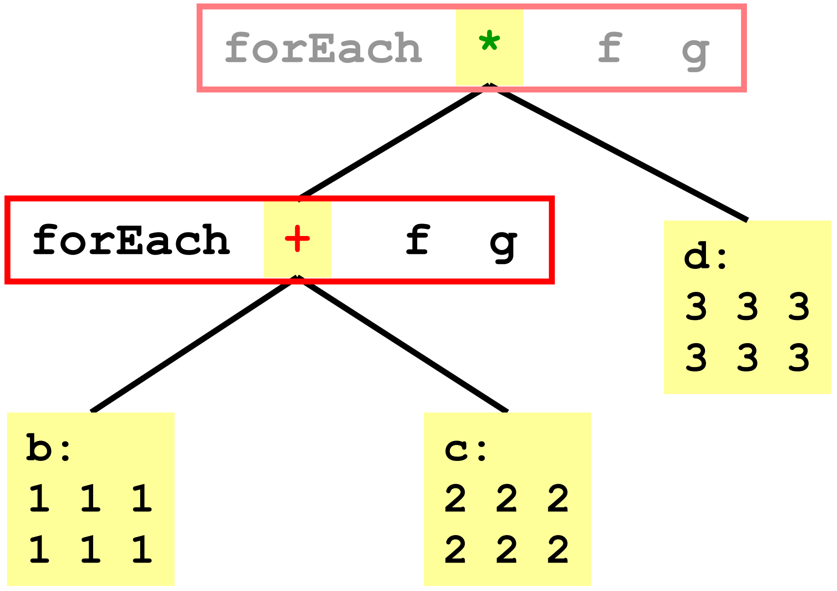


We are at a node:
multiply left and right son

```
-> evaluate left branch
-> evaluate right branch
-> perform multiplication
```

III.4. Expression Templates - Traversing the tree

`f`: (applied at LEAFS) return value of array at index **`i`**
`g`: (applied at NODES) perform operation according to node

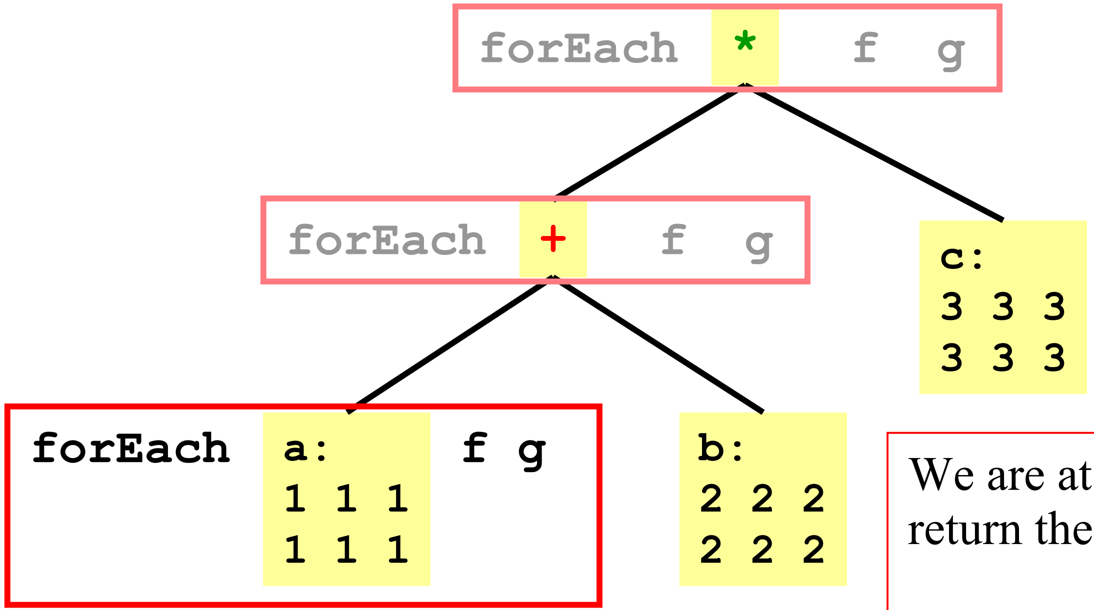


We are at a node:
add left and right son

```
-> evaluate left branch
-> evaluate right branch
-> perform addition
```

III.4. Expression Templates - Traversing the tree

f: (applied to LEAFS) return value of array at index **i**
g: (applied to NODES) perform operation according to node

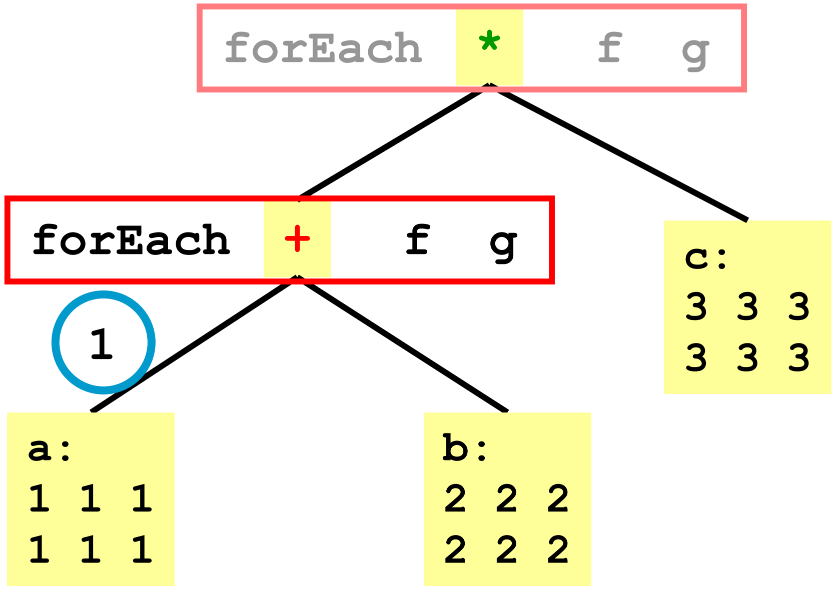


We are at a leaf:
return the **i**’th element of the array

```
-> return a[1]
```

III.4. Expression Templates - Traversing the tree

f: (applied to LEAFS) return value of array at index **i**
g: (applied to NODES) perform operation according to node

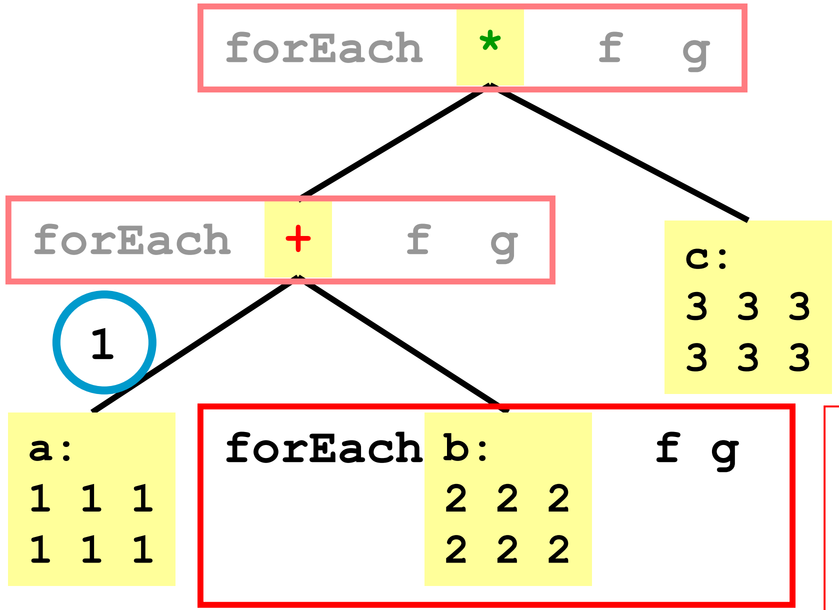


We are at a node:
add left and right son

```
-> evaluate right branch
-> perform addition
```

III.4. Expression Templates - Traversing the tree

f: (applied to LEAFS) return value of array at index **i**
g: (applied to NODES) perform operation according to node

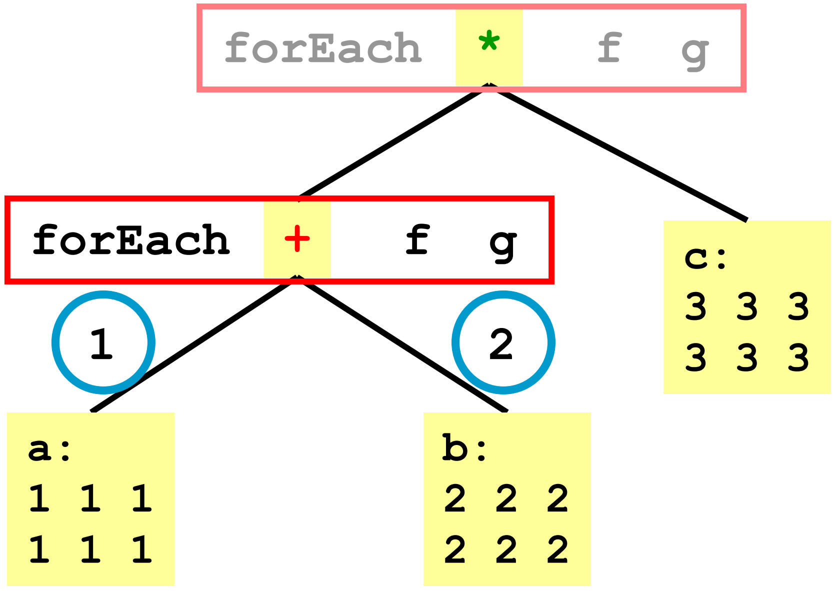


We are at a leaf:
return the **i**'th element of the array

`-> return b[1]`

III.4. Expression Templates - Traversing the tree

f: (applied to LEAFS) return value of array at index **i**
g: (applied to NODES) perform operation according to node

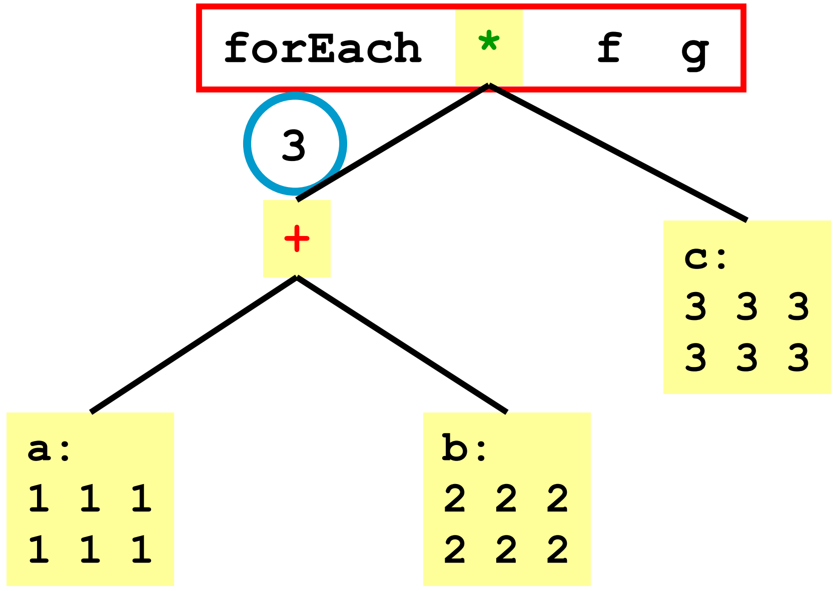


We are at a node:
add left and right son

`-> perform addition`

III.4. Expression Templates - Traversing the tree

`f`: (applied to LEAFS) return value of array at index `i`
`g`: (applied to NODES) perform operation according to node

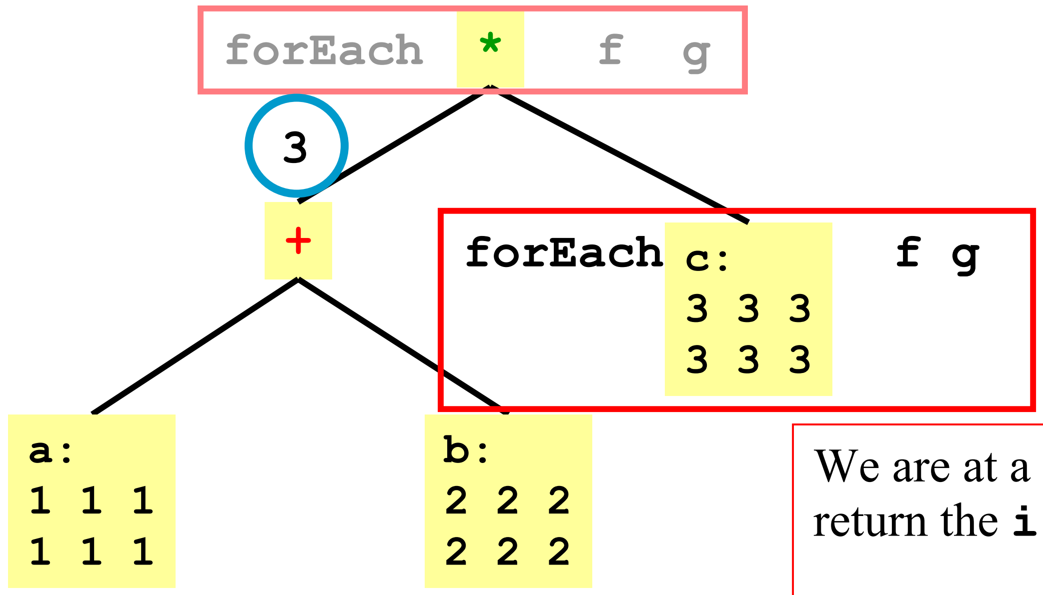


We are at a node:
multiply left and right son

```
-> evaluate right branch
-> perform multiplication
```

III.4. Expression Templates - Traversing the tree

`f`: (applied to LEAFS) return value of array at index **`i`**
`g`: (applied to NODES) perform operation according to node

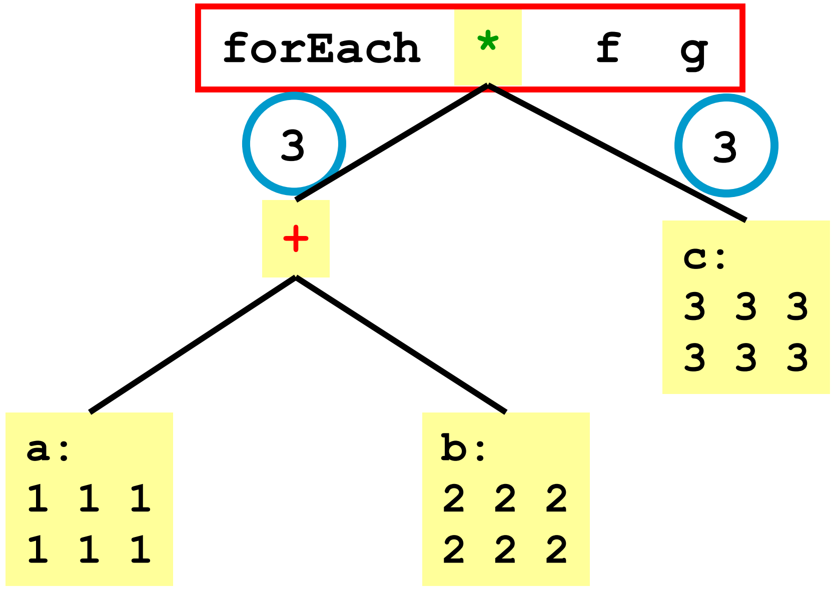


We are at a leaf:
return the **`i`**'th element of the array

```
-> return c[1]
```

III.4. Expression Templates - Traversing the tree

f: (applied to LEAFS) return value of array at index **i**
g: (applied to NODES) perform operation according to node

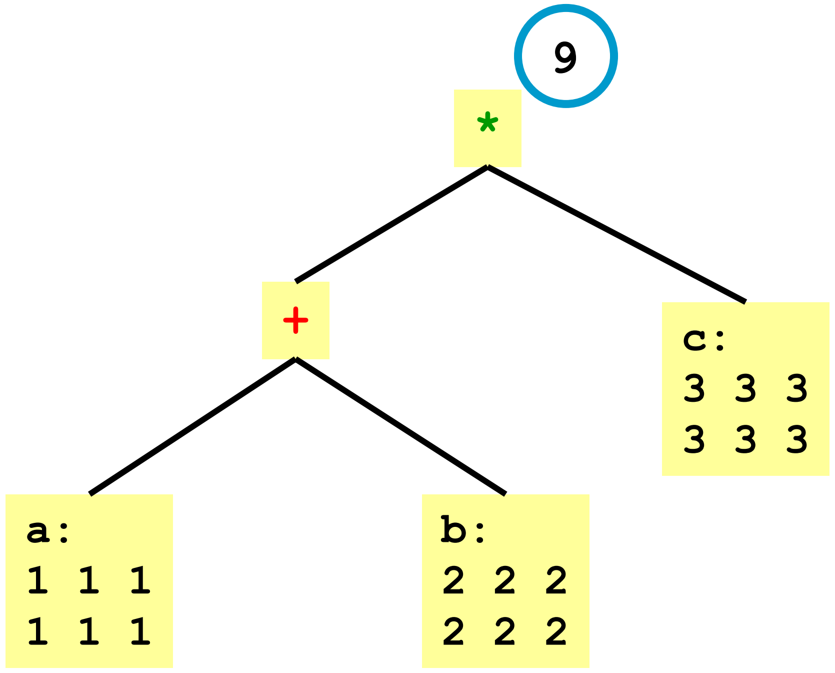


We are at a node:
multiply left and right son

`-> perform multiplication`

III.4. Expression Templates - Traversing the tree

f: (applied to LEAFS) return value of array at index **i**
g: (applied to NODES) perform operation according to node



III.4. Expression Templates - Traversing the tree

- Use template metafunctions to traverse the expression tree and to produce code
- Guiding Code Production as we have done with lists (see **III.3.**)

General Version of **`ForEach`**:

```
template <typename Expression,typename fTag,typename gTag>
struct ForEach;
```

III.4. Expression Templates - Traversing the tree

• When visiting a leaf we perform the action according to the tag `fTag`:

```
template <typename Expression,typename fTag,typename gTag>
struct ForEach {
  typedef LeafFunctor<Expression,fTag>::Type_t Type_t;
  static inline Leaf_t apply(const Expression& e,
                             const fTag& f,
                             const gTag&) {
    return LeafFunctor<Expression,fTag>::apply(e,f);
  }
};
```

• For user-defined types, we need a specialization of **`LeafFunctor`** !

III.4. Expression Templates - Traversing the tree

Example:

```
template <int size>
struct UArray {
  ...
};

struct EvalAt {
   EvalAt(int i) : n(i) {}
   const int& at() const { return n; }
   int n;
};

template <int size>
struct LeafFunctor<UArray<size>, EvalAt > {
   typedef double Type_t;
   static inline
   Type_t apply(const UArray<size>& u,const EvalAt& a) {
      return u[a.at()];
   }
};
```

III.4. Expression Templates - Traversing the tree

• Reaching a binary node, we have to

- `evaluate the left child`
- `evaluate the right child`
- `perform the operation according to gTag`

```
template <class Op,class L,class R,class fTag,class gTag>
struct ForEach<BinaryNode<Op,L,R>, fTag, gTag> {
  typedef typename ForEach<L,fTag,gTag>::Type_t          NL_t;
  typedef typename ForEach<R,fTag,gTag>::Type_t          NR_t;
  typedef typename Combine2<NL_t,NR_t, Op, gTag>::Type_t Type_t;

  static inline
  Type_t apply(const BinaryNode<Op,L,R>& e,
               const fTag& f, const gTag& g) {
    return Combine2<NL_t, NR_t, Op, gTag>::apply(
                ForEach<L,fTag,gTag>::apply(e.left(),f,g),
                ForEach<R,fTag,gTag>::apply(e.right(),f,g),
                e.operation(),
                g );
  }
};
```

III.4. Expression Templates - Traversing the tree

- PETE provides the pre-defined `OpCombine` tag:

```
template <typename A,typename B,typename OP,typename gTag>
struct Combine2 {};

template <typename A,typename B,typename OP>
struct Combine2<A,B,OP, OpCombine> {

  typedef typename BinaryReturn<A,B,OP>::Type_t Type_t;

  static inline
  Type_t apply(A a,B b,OP op,OpCombine) {
    return op(a,b);
  }
};
```

III.4. Expression Templates - How it works

```
UArray a,b,c,d;
a = ((b + c) * d);
```

Resulting Expression Template Tree

```
Expression< BinaryNode< OpMultiply,
                        BinaryNode< OpAdd
                                    UArray,
                                    UArray
                                  >,
                        UArray
                      >
          >
```

- Traverse tree **`size`** times and perform actions according to tree nodes

```
template <typename E>
UArray& UArray::operator=(const Expression<E>& t) {
  for (int i=0 ; i < size; i++)
    data[i] = forEach(t, EvalAt(i), OpCombine() );
}
```

Transformed during compile time !

```
(b[i] + c[i]) * d[i]
```

III.4. Expression Templates - How it works

```
UArray a,b,c,d;
a = ((b + c) * d);
```

Transformed during compile time

```
for (int i=0 ; i<size; i++)
  a[i] = (b[i] + c[i]) * d[i];
```

Single loop !
No temporaries !

No overhead ! ... If using a highly optimizing C++ compiler ...

III.4. Expression Templates - Using PETE (Summary)

To make a user-defined type **`T`** aware of expression templates:

- The user has to provide a specialization for **`CreateLeaf<T>`**
- **`T`** should provide an assignment operator (**`operator=`**) from **`Expression`**
- The user eventually needs to provide a specialization of **`LeafFunctor`**

Who creates all the overloaded operators (e.g. **`operator+`**, **`operator*`**) for the user defined type ?

Use PETE's **`MakeOperators`** tool

`http://www.acl.lanl.gov/pete`

Overview

- Template Metaprogramming & Data Structures
- Type Checking
- Guiding Code Production with Data Structures
- Expression Templates and PETE
- *Advanced Metaprograms*
- References

III.5. Advanced Metaprograms - Checking for Convertibility

How to check whether a type **`A`** could be converted into a type **`B`** ?

Idea: *Use function overloading:*

- **Everything** could be passed to a function that makes use of the **ellipsis**

```
void check(...);
```

- Overlaod **`check`** with a function that expects a value of type **`B`**

```
void check(B dummy);
```

Problem: How to determine which variant has been chosen ?

Idea: **`sizeof`** is a very powerful tool; it can be applied to very complex expressions !

III.5. Advanced Metaprograms - Checking for Convertibility

```
// Metaprogram that check whether an instance of A could be
// converted into an instance of type B

template <typename A,typename B>
struct convertible {
  struct isConvertible    { char a;                 };
  struct isNOTConvertible { char a[sizeof(char)+2];};

  static isConvertible    check(B);
  static isNOTConvertible check(...);

  static A MakeA();

  enum { Ret = ( sizeof(check(MakeA()) == sizeof(isConvertible) ) };
};
```

Avoid call to eventually non existing default constructor !

Notice, that neither **`MakeA`**, nor **`isConvertible`** / **`isNOTConvertible`** must be defined !

III.5. Advanced Metaprograms - Checking for Inheritance

How to check whether a class **`A`** is a subclass of **`B`** ?

Remember what we have said in section **I.2.3:**

10. Pointer conversion **Conversion**

- An rvalue of type **pointer to *cv* D**, where D is a class type, **can be converted to an** rvalue of **type pointer to *cv* B**, **where B is a base class of D**.
- An rvalue of type pointer to *cv* **`T`**, where **`T`** is an object type, can be converted to an rvalue of type pointer to *cv* **`void`**.

```
template <typename A,typename B>
struct SUB_SUPERCLASS {
  enum { Ret =(convertible<const A*,const B*>::Ret &&
               ! EQUAL<const A*,const void*>::Ret ) };
};
```

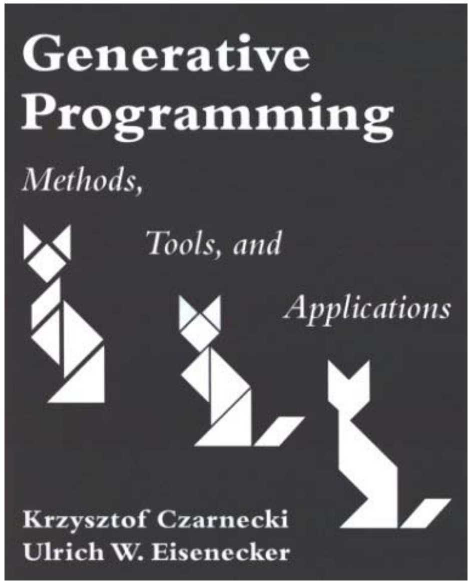
Note: Using this definition **`SUB_SUPERCLASS<A,A>::Ret`** evaluates to **`true`**

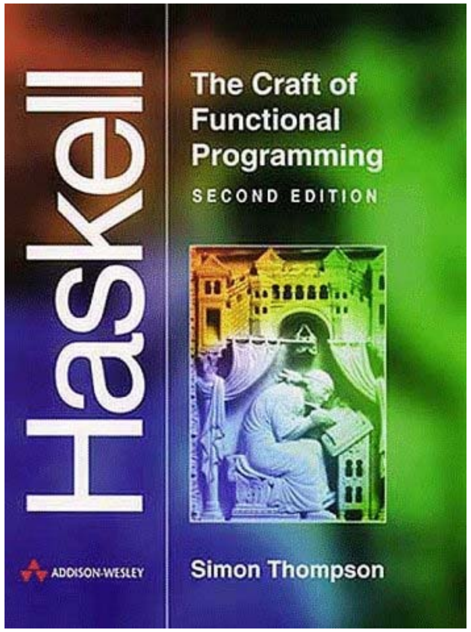
Overview

- Template Metaprogramming & Data Structures
- Type Checking
- Guiding Code Production with Data Structures
- Expression Templates and PETE
- Advanced Metaprograms
- *References*

III.6. References

- *Czarnecki, Eisenecker*: **Generative Programming: Methods, Techniques, and Applications**, Addison-Wesley, 2000





- *S. Thompson*: **Haskell: The Craft of Functional Programming**, Addison-Wesley, 1999

- For more info on Haskell go to **`http://www.haskell.org`**

III.6. References - Some Papers

- *T. Veldhuizen:* **Expression Templates**, C++ Report June 1995
- *T. Veldhuizen*, **Using C++ template metaprograms**, C++ Report May 1995
- *J. Siek, A. Lumsdaine:* **Concept Checking: Binding Parametric Polymorphism in C++**, Proceedings of TMPW 2000 `http://oonumerics.org/tmpw00`
- *J. Striegnitz, S. Smith*: **An Expression Template aware Lambda Function**, Proceedings of TMPW 2000 `http://oonumerics.org/tmpw00`

IV. Some Libraries

PETE

Easy addition of expression template functionality to user-defined classes
`http://www.acl.lanl.gov/pete`

POOMA

Efficient arrays for C++, array statements are executed data-parallel (MPI/threads - based on expression templates)
`http://www.acl.lanl.gov/pooma`

Blitz++

Efficient arrays for C++ - based on expression templates
`http://oonumerics.org/blitz`

FACT!

Functional programming with C++ (runtime - based on expression templates)
`http://www.fz-juelich.de/zam/FACT`

More at **`http://oonumerics.org`**

V. Conferences and Workshops

- European Conference on Object-Oriented Programming (**ECOOP**)
- Conference on Object-Oriented Programming, Systems, Languages, and Applications (**OOPSLA**)
- International Symposium on Computing in Object-Oriented Parallel Environments (**ISCOPE**)
- Workshop on Parallel/High Performance Object-Oriented Scientific Computing (**POOSC**)
- Workshop on C++ Template Programming (**TMPW**)
- Multi-Paradigm Programming with Object-Oriented Languages (**MPOOL**)