C++ Template Metaprogramming

Getting beyond the Container-of-T Stage

Christian Dietrich

Friedrich-Alexander Universität Erlangen-Nürnberg

June 11, 2013

Metaprogramming? What and Why?

- Metaprogramming is a program about a program. It **manipulates code**. The C++ compiler manipulates C++ to assembly.
- C++ templates are a **metaprogamming language** within the language itself, in contrast to an external tool (e.g. a compiler).

Metaprogramming? What and Why?

- Metaprogramming is a program about a program. It **manipulates code**. The C++ compiler manipulates C++ to assembly.
- C++ templates are a **metaprogamming language** within the language itself, in contrast to an external tool (e.g. a compiler).

- Templates can implement domain specific languages within the language itself. **No external tools** are needed.
- Templates interact with and within the C++ type system.
- Templates help you to **fight boilerplate** code.

Metaprogramming? What and Why?

- Metaprogramming is a program about a program. It **manipulates code**. The C++ compiler manipulates C++ to assembly.
- C++ templates are a **metaprogamming language** within the language itself, in contrast to an external tool (e.g. a compiler).

- Templates can implement domain specific languages within the language itself. **No external tools** are needed.
- Templates interact with and within the C++ type system.
- Templates help you to **fight boilerplate** code.

- Our goal for today is a template for defining abstract factories: `AbstractFactory<TYPELIST_3(Button, Window, Scrollbar)>`

Outline

1. Motivation

2. Techniques
   - Integrals and Types as Objects
   - Functions, Conditionals and Repetition
   - Complex Data Structure: Type Lists

3. Case Study: Abstract Factory
   - Abstract Factory Revisited
   - The AbstractFactory's interface
   - Generating Structures

4. Conclusion

Integrals and Types as Objects (I)

Integrals

```
static const int  a = 10;
static const bool b = true;
enum {            c = 30};
```

Integrals and Types as Objects (I)

Integrals

```
static const int  a = 10;
static const bool b = true;
enum {            c = 30};
```

Types (Declaration)

```
struct NullPtr; // No implementation!
struct fibonacci {
  enum { n_1 = 1, n_2 = 1};
}
std::cout << (fibonacci::n_1 + fibonacci::n_2);
```

Integrals and Types as Objects (II)

Types (Assignment)

```
struct constant_pair {
  typedef int     first;  // first  := int
  typedef NullPtr second; // second := NullPtr
};
```

Integrals and Types as Objects (II)

Types (Assignment)

```
struct constant_pair {
  typedef int      first;  // first  := int
  typedef NullPtr second; // second := NullPtr
};
```

Types are Immutable

```
typedef NullPtr       myVariable;
typedef constant_pair myVariable;
```

Integrals and Types as Objects (II)

Types (Assignment)

```
struct constant_pair {
  typedef int     first;  // first  := int
  typedef NullPtr second; // second := NullPtr
};
```

Types are Immutable

```
typedef NullPtr       myVariable;
```

Is already defined here!

~~typedef constant_pair myVariable;~~

Integrals and Types as Objects (II)

Types (

```
struc
   ty
   ty
};
```

Types a

```
typede
~~typede~~
```

- Constant Integral Types are Basic Objects

10

Integrals and Types as Objects (II)

Types (

```
struc
   typ
   typ
};
```

Types a

```
typede
```

~~typede~~

- Constant Integral Types are Basic Objects
- Types are Basic Objects

10 NullPtr

Integrals and Types as Objects (II)

Types (

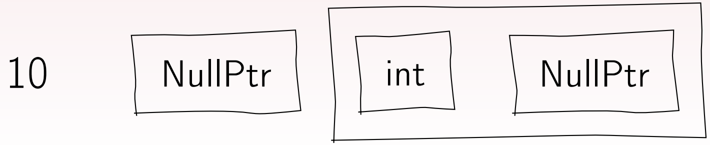
```
struc
   typ
   typ
};
```

Types a

```
typede
```

~~typede~~

- Constant Integral Types are Basic Objects
- Types are Basic Objects
- Types can be nested within structs



Integrals and Types as Objects (II)

Types (

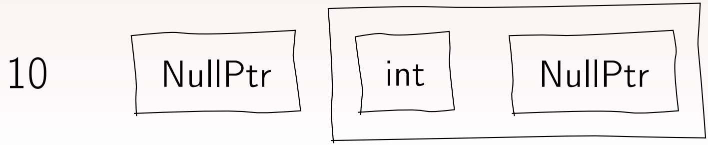
```
struc
   typ
   typ
};
```

Types a

```
typede
```

~~typede~~

- Constant Integral Types are Basic Objects
- Types are Basic Objects
- Types can be nested within structs
- Types are immutable



Functions (I)

Numeric Functions

```
template <int a, int b> struct add_ {
    enum { value = a + b };
};
std::cout << add_<1,2>::value;
```

Functions (I)

parameter list

Numeric Functions

```
template <int a, int b> struct add_ {
    enum { value = a + b };
};
std::cout << add_<1,2>::value;
```

Functions (I)

parameter list name

Numeric Functions

```
template <int a, int b> struct add_ {
    enum { value = a + b };
};
std::cout << add_<1,2>::value;
```

Functions (I)

parameter list

name

Numeric Functions

```
template <int a, int b> struct add_ {
    enum { value = a + b };
};
std::cout << add_<1,2>::value;
```

body

Functions (I)

Numeric Functions

```
template <int a, int b> struct add_ {
    enum { value = a + b };
};
std::cout << add_<1,2>::value;
```

type object

Functions (I)

Numeric Functions

```
template <int a, int b> struct add_ {
    enum { value = a + b };
};
std::cout << add_<1,2>::value;
```

member access

Functions (I)

Numeric Functions

```
template <int a, int b> struct add_ {
    enum { value = a + b };
};
std::cout << add_<1,2>::value;
```

Types as arguments

```
template <typename F, typename S> struct pair_ {
  typedef F first;
  typedef S second;
};
```

Functions (I)

Nur

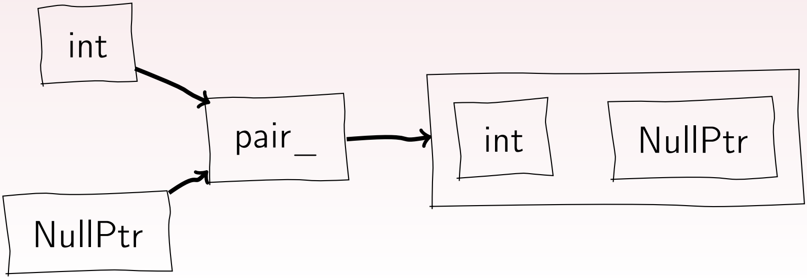
```
tem

};
std
```

Typ

```
tem
  t
  t
};
```

- Templates define Metafunction
- Templates "return" type objects
- `constant_pair` and `pair_<int, NullPtr>` are different objects, but have the same interface



Functions (II)

Prototype

```
template <typename T, typename U>
  struct is_same_type;
```

Implementation

Functions (II)

Prototype

```
template <typename T, typename U>
  struct is_same_type;
```

Implementation

```
template <typename T>
  struct is_same_type<T,T> {
    enum { value = true };
};
```

Functions (II)

Prototype

```
template <typename T, typename U>
  struct is_same_type;
```

Implementation

```
template <typename T>
  struct is_same_type<T,T> {
    enum { value = true };
};

template <typename T, typename U>
  struct is_same_type {
    enum { value = false };
};
```

Conditionals

Prototype

```
template <bool C, typename Then, typename Else >
  struct if_;
```

Implementation

Conditionals

Prototype

```
template <bool C, typename Then, typename Else >
  struct if_;
```

Implementation

```
template <typename Then, typename Else >
  struct if_ <true, Then, Else> {
     typedef Then result;
};
```

Conditionals

Prototype

```
template <bool C, typename Then, typename Else >
  struct if_;
```

Implementation

```
template <typename Then, typename Else >
  struct if_ <true, Then, Else> {
     typedef Then result;
};
```

Conditionals

Prototype

```
template <bool C, typename Then, typename Else >
  struct if_;
```

Implementation

```
template <typename Then, typename Else >
  struct if_ <true, Then, Else> {
     typedef Then result;
};
template <bool C, typename Then, typename Else >
  struct if_ {
     typedef Else result;
};
```

Conditionals

Prototype

```
template <bool C, typename Then, typename Else >
  struct if_;
```

Implementation

```
template <typename Then, typename Else >
  struct if_ <true, Then, Else> {
     typedef Then result;
};
template <bool C, typename Then, typename Else >
  struct if_ {
     typedef Else result;
};
```

Repetition through Recursion

Prototype

```
template <int count> struct fak_;
```

Implementation

Repetition through Recursion

Prototype

```
template <int count> struct fak_;
```

Implementation

```
template <> struct fak_<0> {
      enum { result = 1 };
};
```

Repetition through Recursion

Prototype

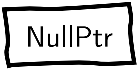
```
template <int count> struct fak_;
```

Implementation

```
template <> struct fak_<0> {
      enum { result = 1 };
};

template <int count> struct fak_ {
  private:
      typedef fak_<count-1> recursion;
  public:
      enum { result = count * recursion::result };
};
```

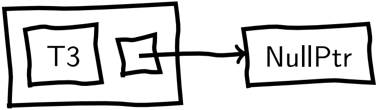
Type Lists



A Typelist

```
typedef
  pair_< T1,
    pair_< T2,
      pair_< T3,
           NullPtr > > >  myTypeList;
```

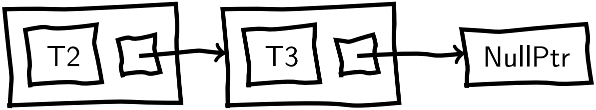
Type Lists



A Typelist

```
typedef
  pair_< T1,
    pair_< T2,
      pair_< T3,
           NullPtr > > >  myTypeList;
```

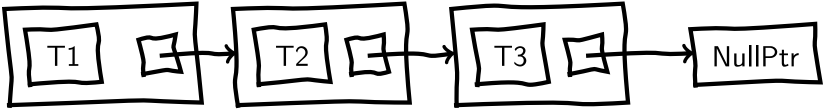
Type Lists



A Typelist

```
typedef
  pair_< T1,
    pair_< T2,
      pair_< T3,
           NullPtr > > >  myTypeList;
```

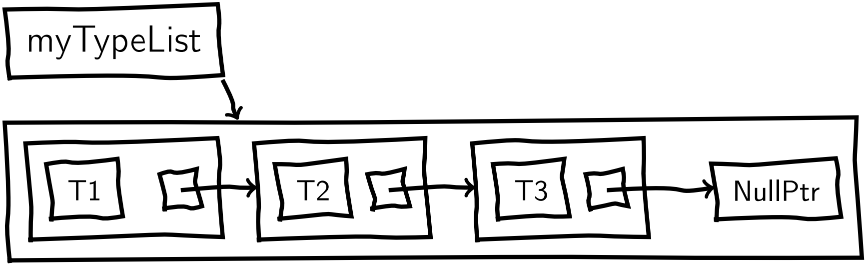
Type Lists



A Typelist

```
typedef
  pair_< T1,
    pair_< T2,
      pair_< T3,
           NullPtr > > >  myTypeList;
```

Type Lists



A Typelist

```
typedef
  pair_< T1,
    pair_< T2,
      pair_< T3,
           NullPtr > > >  myTypeList;
```

Operations on a Type List

Prototype

```
template <class TL> struct length_;
```

Implementation

Operations on a Type List

Prototype

```
template <class TL> struct length_;
```

Implementation

```
template <> struct length_<NullPtr> {
    enum { value = 0 };
};
```

Operations on a Type List

Prototype

```
template <class TL> struct length_;
```

Implementation

```
template <> struct length_<NullPtr> {
    enum { value = 0 };
};

template <class H, class T>
struct length_< pair_<H, T> > {
    enum { value = 1 + length_<T>::value };
};
```

Cost of operations

Usage of length_<TL>

```
std::cout << length_<myTypeList>::value;
```

clang -O3

```
movl $0x3,0x4(%esp)
movl $0x8049cc0,(%esp) // &std::cout
call 8048500 <std::ostream::operator<<(int)@plt>
```

Cost of operations

Usage of length_<TL>

```
std::cout << length_<myTypeList>::value;
```

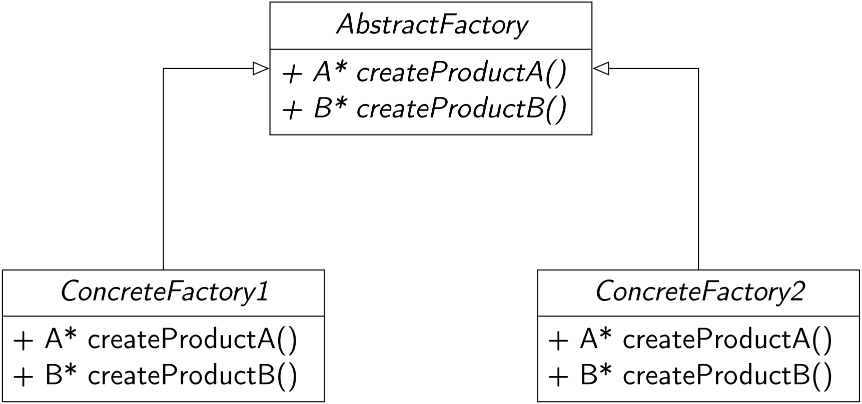
clang -O3

```
movl $0x3,0x4(%esp)
movl $0x8049cc0,(%esp) // &std::cout
call 8048500 <std::ostream::operator<<(int)@plt>
```

Outline

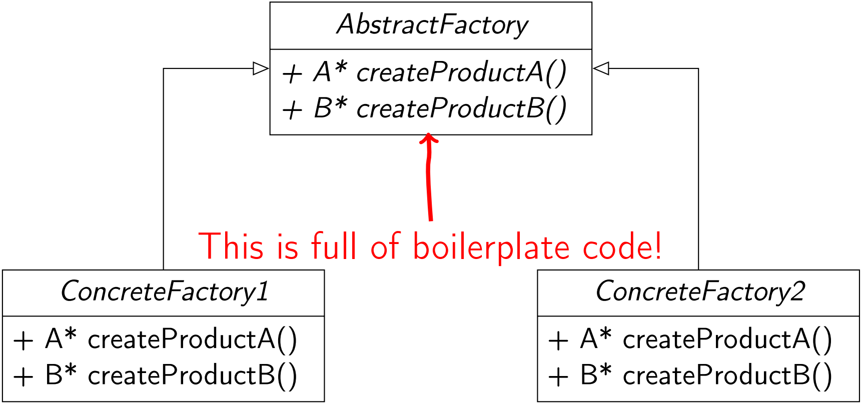
1. Motivation
2. Techniques
   - Integrals and Types as Objects
   - Functions, Conditionals and Repetition
   - Complex Data Structure: Type Lists
3. Case Study: Abstract Factory
   - Abstract Factory Revisited
   - The AbstractFactory's interface
   - Generating Structures
4. Conclusion

Abstract Factory Revisited



> *"Provide an interface for creating families of related or dependent objects without specifying their concrete classes.[3]"*

Abstract Factory Revisited



> *Provide an interface for creating families of related or dependent objects without specifying their concrete classes.[3]*

The AbstractFactory's interface

The AbstractFactory's interface

Definition of an Abstract Factory

```
typedef pair_<int,
          pair_<double,
            pair_<float, NullPtr> > > myTL;
typedef AbstractFactory<myTL> NumberFactory;
```

The AbstractFactory's interface

Definition of an Abstract Factory

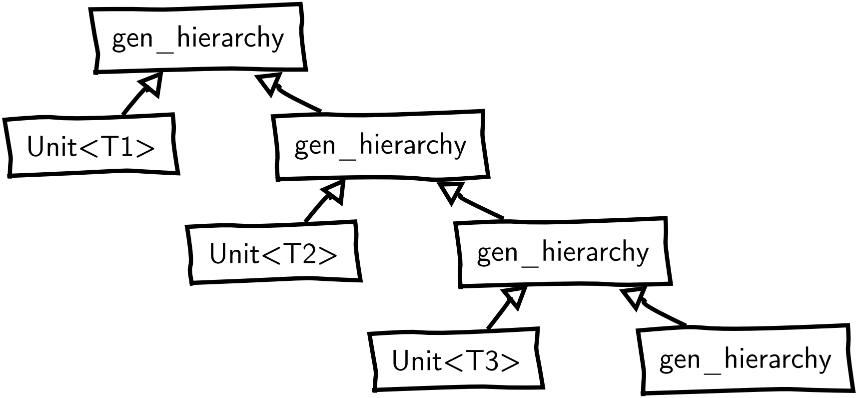
```
typedef pair_<int,
          pair_<double,
             pair_<float, NullPtr> > > myTL;
typedef AbstractFactory<myTL> NumberFactory;
```

Usage of the Factory

```
NumberFactory *factory = ...
int      * a = factory->Create<int>();
double * b = factory->Create<double>();
```

Generating Structures

```
typedef gen_hierarchy<myTypeList, Unit> dummy;
```



Generating Inheritance Hierarchies

Prototype

```
template <typename TL, template <class> class Unit>
  struct gen_hierarchy;
```

Implementation

Generating Inheritance Hierarchies

Prototype

```
template <typename TL, template <class> class Unit>
  struct gen_hierarchy;
```

Implementation

```
template <template <class> class Unit>
  struct gen_hierarchy<NullType, Unit> { };
```

Generating Inheritance Hierarchies

Prototype

```
template <typename TL, template <class> class Unit>
  struct gen_hierarchy;
```

Implementation

```
template <template <class> class Unit>
  struct gen_hierarchy<NullType, Unit> { };

template <typename H, typename T,
          template <class> class Unit>
struct gen_hierarchy<pair_<H, T>, Unit>
   : Unit<H>, gen_hierarchy<T, Unit> {};
```

Filling in the missing bits

The Abstract Factory Unit

```
template <typename T> struct type_ {};
template <typename T> struct AFUnit {
    virtual T * DoCreate(type_<T>) = 0;
};
```

Filling in the missing bits

The Abstract Factory Unit

```
template <typename T> struct type_ {};
template <typename T> struct AFUnit {
    virtual T * DoCreate(type_<T>) = 0;
};
```

The Abstract Factory Template

```
template <typename TL,
          template <class> class Unit = AFUnit>
struct AbstractFactory : gen_hierarchy<TL, Unit> {
    template <class T> T * Create() {
        Unit<T> &u = *this;
        return u.DoCreate(type_<T>());
    } };
```

Outline

1. Motivation
2. Techniques
   - Integrals and Types as Objects
   - Functions, Conditionals and Repetition
   - Complex Data Structure: Type Lists
3. Case Study: Abstract Factory
   - Abstract Factory Revisited
   - The AbstractFactory's interface
   - Generating Structures
4. Conclusion

Conclusion

- Templates are a real programming language
- They use the compiler as their host machine
- They can be used for replacing boilerplate
- Read “Modern C++ Design” by Andrei Alexandrescu!

References I

David Abrahams and Aleksey Gurtovoy.
C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series).
Addison-Wesley Professional, 2004.

Andrei Alexandrescu.
Modern C++ design: generic programming and design patterns applied.
Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.
Design Patterns: Elements of Reusable Object-Oriented Software.
Addison-Wesley Professional, 1 edition, November 1994.