Generic programming and library development

Topics today:

Tuples (and type mappings)

Sources:

- [Vandevoorde and Josuttis 2003] §21 (see also §19, §13.10, and §15.2)
- [Karlsson 2005] §8
- [Jaakko Järvi, Tuple types and multiple return values, *C/C++ Users Journal* **19**,8 (2001), 24–35]

Course home page:

`http://www.diku.dk/forskning/performance-engineering/Generic-programming/`

`pair` and `tuple`

"A tuple is a fixed-size collection of values of specific types."

Often used to group related return values together, or to pass a varying number of arguments into a function template.

You could use `struct` or `class`, ikk'

But the C++ standard library has `std::pair` and the Boost library has `boost::tuple`.

```
tyr> cat set.c++
#include <set> // defines std::set
#include <iostream> // defines standard streams

int main(int, char**) {
  typedef std::set<double> set;
  typedef set::iterator iterator;
  set d;
  std::pair<iterator, bool> handle = d.insert(7.7);
  if (handle.second) {
    std::cout << "inserted" << std::endl;
  }
  else {
    std::cout << "is in already" << std::endl;
  }
  return 0;
}
tyr>~/Script/Gfilt/gfilt -Wall -pedantic -x c++ set.c++
tyr> ./a.out
inserted
```

`struct` and `class`

If you write

```
struct empty {
};
```

or

```
class empty {
};
```

the compiler will give you

```
class empty {
public:
  empty();                   // default constructor
  empty(empty const&);       // copy constructor
  ~empty();                  // destructor
  empty&
  operator=(empty const&);// assignment operator
  empty*
  operator&();               // address-of operator (non-const)
  empty const*
  operator&() const;         // address-of operator (const)
};
```

and allow you to write the following:

```
empty const e;
empty f(e);
f = e;
empty* q = &f;
empty const* p = &e;
```

Tuples in Python

```
tyr> cat tuples.py

k = ('Kenny', 21413)
j = ('Jyrki', 21416)

t = (k, j)

if k < j:
  pass

list = [k, j]
dictionary = {}
dictionary[k[0]] = k[1]
dictionary[j[0]] = j[1]

(name, telephone_number) = k
print t[1]

tyr> python tuples.py
('Jyrki', 21416)
```

`pair` in the CPH STL—oho

```
tyr> cat tuples-cphstl.c++
#include <string> // defines std::string
#include "cphstl/utility" // defines cphstl::pair

int main(int, char**) {
  typedef cphstl::pair<std::string, int> pair;
  pair k = cphstl::make_pair("Kenny", 21413);
  return 0;
}

tyr> make cph
cphstl/utility.c++: In constructor
   ‘cphstl::pair<char[6], int>::pair(const char[6]&, const int &)’:
cphstl/utility.c++:161:   instantiated from
   ‘cphstl::pair<char[6], int>
    cphstl::make_pair(const char[6]&, const int &)’
tuples-cphstl.c++:7:   instantiated from here
cphstl/utility.c++:113: error: ISO C++ forbids
assignment of arrays

tyr> cat makefile
gfilt-dir = $(HOME)/Script/Gfilt
options = -Wall -pedantic -x c++

cph:
$(gfilt-dir)/gfilt $(options) tuples-cphstl.c++
./a.out
```

Problem and its correction

```
template <typename T, typename U>
cphstl::pair<T, U>
make_pair(T const& x, U const& y) {
  return cphstl::pair<T, U>(x, y);
}
```

⟹ [Vandevoorde and Josuttis 2003] footnote on p. 59

```
template <typename T, typename U>
cphstl::pair<T, U>
make_pair(T x, U y) {
  return cphstl::pair<T, U>(x, y);
}
```

It helps, but why?

Next problem

```
tyr> cat tuples-cphstl.c++
#include <string> // defines std::string
#include "cphstl/utility" // defines cphstl::pair
#include <iostream> // defines standard streams
#include <list> // defines std::list
#include <map> // defines std::map

int main(int, char**) {
  typedef cphstl::pair<std::string, int> pair;
  pair k("Kenny", 21413);
  pair j = cphstl::make_pair("Jyrki", 21416);

  typedef cphstl::pair<pair, pair> pair_of_pairs;
  pair_of_pairs t = cphstl::make_pair(k, j);

  if (k < j) {}

  pair a[] = {k, j};
  std::list<pair> list(&a[0], &a[sizeof(a)/sizeof(pair)]);

  std::map<std::string, int> dictionary;
  dictionary.insert(k);
  dictionary.insert(cphstl::make_pair("Jyrki", 21416));
  return 0;
}

BD Software STL Message Decryptor v2.47a for gcc
tuples-cphstl.c++: In function ‘int main(int, char **)’:
tuples-cphstl.c++:23: error: No match for
   ‘map<string, int>::insert(main(int, char **)::pair &)’
tuples-cphstl.c++:24: error: No match for
   ‘map<string, int>::insert(cphstl::pair<const char *, int>)’
```

Problem and its correction

The `std::map` can only store `std::pairs` so we must provide a conversion function which converts `cphstl::pairs` to `std::pairs`.

```
namespace cphstl {

  template <typename T, typename U>
  class pair {
  public:
    ...

    template <typename V, typename W>
    operator std::pair<V, W>() const {
      return std::pair<V, W>((*this).first, (*this).second);
    }
   ...
```

Online exercise

Assume that I add `operator==` as a member function to the `cphstl::pair` class template. When a `std::pair` and a `cphstl::pair` are compared, which `operator==` is called?

```
bool operator==(pair const& other) const {
  return (*this).first == other.first
         && (*this).second == other.second;
}
```

Next challenge

```
...
std::string name;
int telephone_number;
cphstl::tie(name, telephone_number) = k;
...
```

```
BD Software STL Message Decryptor v2.47a for gcc
cphstl/utility.c++: In instantiation of
‘cphstl::pair<string &, int &>’:
tuples-cphstl.c++:29:   instantiated from here
cphstl/utility.c++:115: error: forming reference
to reference type ‘int &’
tuples-cphstl.c++: In member function
   ‘cphstl::pair<string &, int &> &
    cphstl::pair<string &, int &>::operator=(
        const cphstl::pair<string &, int &> &
    )’:
...
```

Problem and its correction

```
pair(T const& x, U const& y)
  : first(x), second(y) {
}
```

$\Longrightarrow$

Use `boost:tuples` instead! Do not try to invent the wheel again.

Summary: `pair`s

Here is the list of goodies provided by `cphstl::pair` compared to `struct` or `class`.

- Less typing;
- Constructors zero-initialize data members for built-in types;

```
pair()
  : first(), second() {
}
```

- Implicit type conversions during construction;

```
template <typename V, typename W>
pair (pair<V, W> const& p)
  : first(p.first), second(p.second) {
}
```

- Implicit type conversions during assignments;

```
template <typename V, typename W>
pair<T, U>& operator=(pair<V, W> const& other) {
  first = other.first;
  second = other.second;
  return *this;
}
```

- Comparison operators allow mixed types;

```
template <typename T, typename U, typename V,
  typename W>
bool operator==(pair<T, U> const& x,
                pair<V, W> const& y) {
  return (x.first == y.first)
         && (x.second == y.second);
}
```

- Convenience function `make_pair` so that in construction the types of template arguments are deduced automatically.

`tuple` in the Boost library

```
#include <string> // defines std::string
#include <iostream> // defines standard streams
#include <list> // defines std::list
#include <map> // defines std::map
#include "boost/tuple/tuple.hpp" // library core
#include "boost/tuple/tuple_io.hpp" // I/O operations
#include "boost/tuple/tuple_comparison.hpp" // relational operations

int main(int, char**) {
  typedef boost::tuple<std::string, int> tuple;
  tuple k("Kenny", 21413);
  tuple j = boost::make_tuple("Jyrki", 21416);

  typedef boost::tuple<tuple, tuple> tuple_of_tuples;
  tuple_of_tuples t(k, j);

  if (k < j) {}

  tuple a[] = {k, j};
  std::list<tuple> list(&a[0], &a[sizeof(a)/sizeof(tuple)]);

  std::map<std::string, int> dictionary;
  dictionary[k.get<0>()] = k.get<1>();
  dictionary[j.get<0>()] = j.get<1>();

  std::string name;
  int telephone_number;
  boost::tie(name, telephone_number) = k;

  std::cout << t.get<1>() << std::endl;
  return 0;
}
```

```
tyr> make boo
/home/disk04/jyrki/Script/Gfilt/gfilt -Wall -pedantic -x c++ \
-I /vol/image/disk05/scratch/dadiu/boost_1_33_1 tuples-boost.c++
./a.out
(Jyrki 21416)
```

Summary: `tuples`

- I am really impressed!
- They make the code clearer.
- This library will be part of the C++ standard library.
- Compilation may become a bit slower, but normally this is not a problem.
- The generated code is almost equally fast as a hand-drafted version of the same code.
- As to C++, one should not be judgemental of the language itself, since it was designed to handle its own shortcomings through libraries.
- Be aware that you cannot use tuples containing more than 10 or so fields.

Implementation as recursive pairs

$(x_1, x_2, x_3, x_4, x_5, x_6, x_7, \text{null})$

$\Longrightarrow$

$(x_1, (x_2, (x_3, (x_4, (x_5, (x_6, (x_7, \text{null})))))))$

```
class NullT {
}

Duo<int, Duo<char, Duo<bool, Duo<double, NullT>>>> t;

double d = t.v2().v2().v2().v1();
```

The idea is to use metaprogramming for code generation so that the use of tuples would be more convenient.

Number of fields

```
// partial specialization of the Duo template (where N = 2)
template <typename A, typename B, typename C>
class Duo<A, Duo<B,C> > {
  public:
    typedef A        T1;            // type of first field
    typedef Duo<B,C> T2;            // type of second field
    enum { N = Duo<B,C>::N + 1 };  // number of fields

  private:
    T1 value1;          // value of first field
    T2 value2;          // value of second field

  public:
    // all public members
    ...
};
```

Number of fields (cont.)

```
// partial specialization for Duo<> with only one field
template <typename A>
class Duo<A, NullT> {
  public:
    typedef A     T1;  // type of first field
    typedef NullT T2;  // type of second field
    enum { N = 1 };    // number of fields

  private:
    T1 value1;        // value of first field

  public:
    // constructors
    Duo() : value1() {
    }
    Duo (T1 const & a)
     : value1(a) {
    }

    // field access
    T1& v1() {
        return value1;
    }
    T1 const& v1() const {
        return value1;
    }

    NullT& v2() {
    }
    NullT v2() const {
    }
    //...
};
```

Type of fields

```
// primary template for type of Nth field of (duo) T
template <int N, typename T>
class DuoT {
  public:
    typedef void ResultT; // the general result type
};
// specialization for 1st field of a plain duo
template <typename A, typename B>
class DuoT<1, Duo<A,B> > {
  public:
    typedef A ResultT;
};
// specialization for 2nd field of a plain duo
template <typename A, typename B>
class DuoT<2, Duo<A,B> > {
  public:
    typedef B ResultT;
};
// specialization for Nth field of a recursive duo
template <int N, typename A, typename B, typename C>
class DuoT<N, Duo<A, Duo<B,C> > > {
  public:
    typedef typename DuoT<N-1, Duo<B,C> >::ResultT ResultT;
};
// specialization for 1st field of a recursive duo
template <typename A, typename B, typename C>
class DuoT<1, Duo<A, Duo<B,C> > > {
  public:
    typedef A ResultT;
};
// specialization for 2nd field of a recursive duo
template <typename A, typename B, typename C>
class DuoT<2, Duo<A, Duo<B,C> > > {
  public:
    typedef B ResultT;
};
```

Value of fields

Usage:

`val<3>(t)` generates the code `t.v2().v2()`

```
// return Nth value of variable duo
template <int N, typename A, typename B>
inline
typename
TypeOp<typename DuoT<N, Duo<A, B> >::ResultT>::RefT
val(Duo<A, B>& d) {
  return DuoValue<N, Duo<A, B> >::get(d);
}

// return Nth value of constant duo
template <int N, typename A, typename B>
inline
typename
TypeOp<typename DuoT<N, Duo<A, B> >::ResultT>::RefConstT
val(Duo<A, B> const& d) {
  return DuoValue<N, Duo<A, B> >::get(d);
}
```


Value of fields (cont.)

```
// primary template for value of Nth field of (duo) T
template <int N, typename T>
class DuoValue {
  public:
    static void get(T&) { // in general, no value
    }
    static void get(T const&) {
    }
};
// specialization for 1st field of a plain duo
template <typename A, typename B>
class DuoValue<1, Duo<A, B> > {
  public:
    static A& get(Duo<A, B> &d) {
        return d.v1();
    }
    static A const& get(Duo<A, B> const &d) {
        return d.v1();
    }
};
// specialization for 2nd field of a plain duo
template <typename A, typename B>
class DuoValue<2, Duo<A, B> > {
  public:
    static B& get(Duo<A, B> &d) {
        return d.v2();
    }
    static B const& get(Duo<A, B> const &d) {
        return d.v2();
    }
};
```


Value of fields (cont.)

```
// specialization for Nth field of recursive duo
template <int N, typename A, typename B, typename C>
struct DuoValue<N, Duo<A, Duo<B,C> > > {
    static
    typename
    TypeOp<typename DuoT<N-1, Duo<B,C> >::ResultT>::RefT
    get(Duo<A, Duo<B,C> > &d) {
        return DuoValue<N-1, Duo<B,C> >::get(d.v2());
    }

    static typename TypeOp<typename DuoT<N-1, Duo<B,C>
                              >::ResultT>::RefConstT
    get(Duo<A, Duo<B,C> > const &d) {
        return DuoValue<N-1, Duo<B,C> >::get(d.v2());
    }
};
// specialization for 1st field of recursive duo
template <typename A, typename B, typename C>
class DuoValue<1, Duo<A, Duo<B,C> > > {
  public:
    static A& get(Duo<A, Duo<B,C> > &d) {
        return d.v1();
    }
    static A const& get(Duo<A, Duo<B,C> > const &d) {
        return d.v1();
    }
};
// specialization for 2nd field of recursive duo
template <typename A, typename B, typename C>
class DuoValue<2, Duo<A, Duo<B,C> > > {
  public:
    static B& get(Duo<A, Duo<B,C> > &d) {
        return d.v2().v1();
    }
    static B const& get(Duo<A, Duo<B,C> > const &d) {
        return d.v2().v1();
    }
};
```

Tuple construction

```
// Tuple<> derives from Tuple<> with one more NullT
template <typename P1,
          typename P2 = NullT,
          typename P3 = NullT,
          typename P4 = NullT,
          typename P5 = NullT>
class Tuple
 : public Duo<P1, typename Tuple<P2,P3,P4,P5,NullT>::BaseT> {
  public:
    typedef Duo<P1, typename Tuple<P2,P3,P4,P5,NullT>::BaseT>
            BaseT;

    // constructors:
    Tuple() {}
    Tuple(typename TypeOp<P1>::RefConstT a1,
          typename TypeOp<P2>::RefConstT a2,
          typename TypeOp<P3>::RefConstT a3 = NullT(),
          typename TypeOp<P4>::RefConstT a4 = NullT(),
          typename TypeOp<P5>::RefConstT a5 = NullT())
     : BaseT(a1, Tuple<P2,P3,P4,P5,NullT>(a2,a3,a4,a5)) {
    }
};

// specialization to end deriving recursion
template <typename P1, typename P2>
class Tuple<P1,P2,NullT,NullT,NullT>
 : public Duo<P1,P2> {
  public:
    typedef Duo<P1,P2> BaseT;
    Tuple() {}
    Tuple(typename TypeOp<P1>::RefConstT a1,
          typename TypeOp<P2>::RefConstT a2,
          typename TypeOp<NullT>::RefConstT = NullT(),
          typename TypeOp<NullT>::RefConstT = NullT(),
          typename TypeOp<NullT>::RefConstT = NullT())
     : BaseT(a1, a2) {
    }
};
```

Tuple construction (cont.)

```
// specialization for singletons
template <typename P1>
class Tuple<P1,NullT,NullT,NullT,NullT>
 : public Duo<P1,void> {
  public:
    typedef Duo<P1,void> BaseT;
    Tuple() {}
    Tuple(typename TypeOp<P1>::RefConstT a1,
          typename TypeOp<NullT>::RefConstT = NullT(),
          typename TypeOp<NullT>::RefConstT = NullT(),
          typename TypeOp<NullT>::RefConstT = NullT(),
          typename TypeOp<NullT>::RefConstT = NullT())
     : BaseT(a1) {
    }
};
```

Convenience functions `make_tuple`

```
// convenience function for 1 argument
template <typename T1>
inline
Tuple<T1> make_tuple(T1 const &a1) {
    return Tuple<T1>(a1);
}
// convenience function for 2 arguments
template <typename T1, typename T2>
inline
Tuple<T1,T2> make_tuple(T1 const &a1, T2 const &a2) {
    return Tuple<T1,T2>(a1,a2);
}
// convenience function for 3 arguments
template <typename T1, typename T2, typename T3>
inline
Tuple<T1,T2,T3> make_tuple(T1 const &a1, T2 const &a2,
                           T3 const &a3) {
    return Tuple<T1,T2,T3>(a1,a2,a3);
}
// convenience function for 4 arguments
template <typename T1, typename T2, typename T3, typename T4>
inline
Tuple<T1,T2,T3,T4> make_tuple(T1 const &a1, T2 const &a2,
                              T3 const &a3, T4 const &a4) {
    return Tuple<T1,T2,T3,T4>(a1,a2,a3,a4);
}
// convenience function for 5 arguments
template <typename T1, typename T2, typename T3,
          typename T4, typename T5>
inline
Tuple<T1,T2,T3,T4,T5> make_tuple(T1 const &a1, T2 const &a2,
                                 T3 const &a3, T4 const &a4,
                                 T5 const &a5) {
    return Tuple<T1,T2,T3,T4,T5>(a1,a2,a3,a4,a5);
}
```

Yet another challenge

```
k = ('Kenny', 'Assistant Professor', 21413)

for i in k:
  print i

tyr> python more.py
Kenny
Assistant Professor
21413
```

Tuples and `for_each`

```
#include <iostream>
#include <string>
#include <functional>
#include "boost/tuple/tuple.hpp"

template <typename Function>
void
for_each_element (const boost::tuples::null_type&,
                  Function) {
}

template <typename Tuple, typename Function>
void
for_each_element (Tuple& t, Function func) {
  func(t.get_head());
  for_each_element(t.get_tail(), func);
}

struct print {
  template <typename T>
  void operator() (T const& t) {
    std::cout << t << std::endl;
  }
};

template <typename T>
struct print_type {
  void operator() (T const& t) {
    std::cout << t << std::endl;
  }

  template <typename U>
  void operator() (U const& u) {
  }
};
```

Tuples and `for_each` (cont.)

```
int main() {
  boost::tuple<int, short, double> nums(1, 2, 3.4);
  for_each_element(nums, print());
  for_each_element(nums, print_type<double>());
}
```

Techniques used in compile-time metaprogramming

- Partial specialization to get recursion
- `if_then_else` class template to get conditional branches
- `typedefs` to get assignments
- Discarding overload
- `sizeof`-tricks
- Ellipsis (. . .) construct (utilizing SFINAE)

Compile-time reflection

```
/*
Author: efim.slobodov233432@arcor.de, April 6th, 2005
Source: http://www.codecomments.com/archive324-2005-4-448963.html
*/

typedef char Small;
typedef struct {char unused[2];} Big;

template<int>
class Selector;

template <typename T>
struct Has_member_dual {
private:
  template <typename U>
  static ::Small test(::Selector<sizeof(&U::dual)>*);

  template <typename U>
  static ::Big test(...);

public:
  enum{value=sizeof(test<T>(0))==sizeof(::Small)};
};

class my1 {
};
class my2 {
public:
  void dual();
};

#include <cassert>
int main(int, char **) {
  assert(Has_member_dual<my1>::value == 0);
  assert(Has_member_dual<my2>::value == 1);
  assert(Has_member_dual<int>::value == 0);
  return 0;
}
```

Conclusions

- Template metaprogramming is simple programming with terrible syntax.
- To get the type right can be tricky. Therefore, Bjarne proposed already in the beginning of 80s the `auto` keyword.
- C++ has some shortcomings to support template metaprogramming, e.g. an unbounded number of template parameters (a list or an array of types).

Type mapping:
T $\rightarrow$ reference of T

```
template <typename U>
class reference {
public:
  typedef U& type;
};

template <typename U>
class reference <U&> {
public:
  typedef U& type;
};

template <>
class reference <void> {
public:
  typedef void type;
};
```

Type mapping: T $\rightarrow$ const reference of T

```
template <typename U>
class const_reference {
public:
  typedef U const& type;
};

template <typename U>
class const_reference <U const> {
public:
  typedef U const& type;
};

template <typename U>
class const_reference <U&> {
public:
  typedef U const& type;
};

template <>
class const_reference <void> {
public:
  typedef void type;
};
```

Type mappings in the CPH STL

Usage: Assume that the domain type is `T`.

```
#include "cphstl/type"
typename cphstl::type<T>::reference x;
if (cphstl::type<T>::is_pointer) {}

template <typename T>
class type {
public:
  typedef T original_type;
  typedef ... unqualified;
  typedef ... const_qualified;
  typedef ... volatile_qualified;
  typedef ... const_volatile_qualified;
  typedef ... non_const;
  typedef ... non_volatile;
  typedef ... reference;
  typedef ... const_reference;
  typedef ... pointer;
  typedef ... const_pointer;

  enum {
    is_unsigned_integral = ...,
    is_signed_integral = ...,
    is_other_integral = ...,
    is_integral = ...,
    is_floating_point = ...,
    is_arithmetic = ...,
    is_fundamental = ...,
    is_pointer = ...,
    is_reference = ...;
    is_array = ...,
    is_function = ...,
    is_pointer_to_member = ...,
    is_class = ...
  };
};
```