Copenhagen, 17 December 2001

Title:

New CPH STL headers
`<compile-time-assert>` and `<type>`

Speaker:

Jyrki Katajainen

Datalogisk Institut
Københavns Universitet

My source: Andrei Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley (2001)

Partial template specialization

```
template <typename T>
class X {

  // Most general version.
}

template <typename T>
class X<T*> {

  // Version for general pointers.
}

template <>
class X<void*> {

  // Version for one specific pointer type.
}
```

Local classes

```
void fun() {
  class local {
    ... member variables ...
    ... member function definitions ...
  }
  ... code using local ...
}
```

- Local classes cannot define static member variables and cannot access nonstatic variables.
- Local classes defined inside template functions can use the template parameters of the enclosing function.

Compile-time computations in C++

Operands: typelists, types, or compile-time numeric constants

Operations: `template class`, `typedef`, `enum`, `static constant`, `sizeof`

Restrictions: All compile-time values are **immutable**! After you have defined an integral constant, say an enumerated value, you cannot change it. `typedef`s can be seen as introducing named type constants. Again after definition, they are frozen—you cannot later redefine a `typedef`'d symbol to hold another type.

Although C++ is mostly an imperative language, any compile-time computation must rely on techniques that definitely are reminiscent of pure functional languages—languages unable to mutate values.

Where are these needed?

- doing optimizations (e.g., `copy`)
- handling C++ anomalies (e.g., `pair`)
- implementing concept checks
- implementing tuples (generatizations of pairs)
- implementing integer classes: `integer<8>`, `integer<16>`, `integer<32>`, etc.

Compile-time assertions in C

```
tyr> cat c.c
#define STATIC_CHECK(expression) { \
  char compile_time_assert[(expression) ? 1 : -1]; \
  compile_time_assert[0] = 0; \
}

template <typename to, typename from>
to safe_reinterpret_cast(from source) {
  STATIC_CHECK(sizeof(from) <= sizeof(to));
  to target = reinterpret_cast<to>(source);
  return target;
}

int main() {
  void* p = 0;
  char c = safe_reinterpret_cast<char>(p);
  c = 0;
  void* buf = safe_reinterpret_cast<void*>(0xF00F);
  buf = 0;
  return 0;
}

tyr> g++ -Wall c.c
c.c: In function
‘char safe_reinterpret_cast<char, void *>(void *)’:
c.c:15:   instantiated from here
c.c:8: size of array ‘compile_time_assert’ is negative
c.c:9: reinterpret_cast from ‘void *’ to ‘char’ loses
precision
```

compile-time-assert File Reference

This file defines the macro `compile_time_assert` that performs a compile-time assertion. More...

Go to the source code of this file.

Namespaces

namespace **cphstl**

Compounds

class cphstl::compile_time_checker

Defines

#define compile_time_assert(expression, message)
Macro `compile_time_assert`. More...

Detailed Description

This file defines the macro `compile_time_assert` that performs a compile-time assertion.

>

Generated at Mon Dec 17 01:40:31 2001 for TheCopenhagenSTL by  *1.2.3 written by Dimitri van Heesch, © 1997-2000*

Compile-time assert

Compounds

class cphstl::compile_time_checker

Defines

#define compile_time_assert(expression, message)
Macro `compile_time_assert`. More...

Detailed Description

This module contains the definitions associated with the CPH STL extension `<compile-time-assert>`.

Original author

Jyrki Katajainen <jyrki@diku.dk>, December 2001

Define Documentation

#define compile_time_assert(expression, message)

Initializer:

```
{ \
    struct ERROR_##message { \
    }; \
    typedef cphstl::compile_time_checker<(expression)> type; \
    type temp = type(ERROR_##message()); \
    (void) sizeof(temp); \
  }
```

Macro `compile_time_assert`.

Input

A Boolean expression that can be evaluated at compile time and an error message in the form of a legal C++ identifier (no spaces, should not start with a digit, and so on).

Effect

If the compile-time expression evaluates to `true`, the resulting program is valid. Otherwise a compile-time error occurs.

Implementation

The implementation is taken from the book: Andrei Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied*, Addison-Wesley (2001), see Section 2.1.

---

compile-time-assert

Go to the documentation of this file.

```
00001 /* -*- C++ -*- $Id: compile-time-assert-source.ps,v 1.1 2002/01/22 11:15:3
00002
00019 #ifndef __cphstl_compile_time_assert__
00020 #define __cphstl_compile_time_assert__
00021
00022 namespace cphstl {
00023
00045   template<bool>
00046   class compile_time_checker {
00047   public:
00051     compile_time_checker(...) {
00052     }
00053   };
00054
00055   template<>
00056   class compile_time_checker<false> {
00057   };
00058
00086   #define compile_time_assert(expression, message) { \
00087     struct ERROR_##message { \
00088     }; \
00089     typedef cphstl::compile_time_checker<(expression)> type; \
00090     type temp = type(ERROR_##message()); \
00091     (void) sizeof(temp); \
00092   }
00093 }
00094
00095 #endif // __cphstl_compile_time_assert__
```

Testing it

```
grimer> cat Testsuite/c++.cpp
#include <compile-time-assert>

template <typename to, typename from>
to safe_reinterpret_cast(from source) {
  compile_time_assert(sizeof(from) <= sizeof(to),
    reinterpret_cast_looses_precision);
  to target = reinterpret_cast<to>(source);
  return target;
}

int main() {
  void* p = 0;
  char c = safe_reinterpret_cast<char>(p);
  c = 0;
  void* buf = safe_reinterpret_cast<void*>(0xF00F);
  buf = 0;
  return 0;
}

grimer> gmake unittest
g++ -I. -O6 -Wall  Testsuite/c++.cpp -o Testsuite/c++.exe
Testsuite/c++.cpp: In function
‘char safe_reinterpret_cast<char, void *>(void *)’:
Testsuite/c++.cpp:13:   instantiated from here
Testsuite/c++.cpp:5: no matching function for call to
‘cphstl::compile_time_checker<false>::compile_time_checker
(safe_reinterpret_cast<char, void *>(void *)::
ERROR_reinterpret_cast_looses_precision)’
compile-time-assert:52: candidates are:
cphstl::compile_time_checker<false>::
compile_time_checker(const cphstl::compile_time_checker<fals
compile-time-assert:52:
cphstl::compile_time_checker<false>::compile_time_checker()
Testsuite/c++.cpp:7: reinterpret_cast from ‘void *’ to ‘char
precision
```

type File Reference

This file defines a collection of tools for performing type mappings. More...

Go to the source code of this file.

Namespaces

namespace unnamed
namespace **cphstl**

Compounds

class cphstl::convert
struct cphstl::int2type
struct **cphstl::convert::plain_to_reference**
class cphstl::query
struct **cphstl::test::same**
struct cphstl::select
class cphstl::test
struct cphstl::type2type
struct cphstl::typelist
struct **cphstl::convert::un_const**
struct **cphstl::convert::un_volatile**

Defines

#define TYPELIST_1(T1) typelist<T1, nil>
This macro defines a typelist of lenght 1.

#define TYPELIST_2(T1, T2) typelist<T1, TYPELIST_1(T2) >
This macro defines a typelist of lenght 2.

#define TYPELIST_3(T1, T2, T3) typelist<T1, TYPELIST_2(T2, T3) >
This macro defines a typelist of lenght 3.

#define TYPELIST_4(T1, T2, T3, T4) typelist<T1, TYPELIST_3(T2, T3, T4) >
This macro defines a typelist of lenght 4.

Detailed Description

This file defines a collection of tools for performing type mappings.

Type mappings

Compounds

class	cphstl::convert
struct	cphstl::index_of
struct	cphstl::int2type
struct	cphstl::select
class	cphstl::test
struct	cphstl::type2type
struct	cphstl::typelist

Detailed Description

This module is a CPH STL extension defining a collection of tools for performing type mappings.

Original author

Jyrki Katajainen <jyrki@diku.dk>, December 2001

Sources

Andrei Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied*, Addison-Wesley (2001), see Chapters 2 and 3.

Jaakko Järvi, Tuple types and multiple return values, *C/C++ Users Journal* **19**,8 (2001), 24-35.

type

Go to the documentation of this file.

```
00001 /* -*- C++ -*- $Id: type-source1.ps,v 1.1 2002/01/22 11:15:39 jyrki Exp $
00002
00028 #ifndef __cphstl_type__
00029 #define __cphstl_type__
00030
00031 namespace cphstl {
00032
00038   template <int v>
00039   struct int2type {
00040     enum { value = v };
00041   };
00042
00049   template <typename T>
00050   struct type2type {
00054     typedef T original_type;
00055   };
00056
00061   template <bool, typename T, typename U>
00062   struct select {
00066     typedef T type;
00067   };
00068
00069   template <typename T, typename U>
00070   struct select<false, T, U> {
00071     typedef U type;
00072   };
00073
00078   template <typename T, typename U>
00079   class test {
00080   private:
00084     template <typename X, typename Y>
00085     struct same {
00086       enum { pos = false };
00087       enum { neg = true };
00088     };
00089     template <typename X>
00090     struct same<X, X> {
00091       enum { pos = true };
00092       enum { neg = false };
00093     };
00094
00095   public:
00096     enum { is_equal_to = same<T, U>::pos };
00097     enum { is_not_equal_to = same<T, U>::neg };
00098   };
00099
00104   template <typename T>
00105   class convert {
00106   private:
00110     template <typename U>
00111     struct un_const {
00112       typedef U type;
00113     };
00114     template <typename U>
```

```
00115     struct un_const<const U> {
00116       typedef U type;
00117     };
00118
00122     template <typename U>
00123     struct un_volatile {
00124       typedef U type;
00125     };
00126     template <typename U>
00127     struct un_const<volatile U> {
00128       typedef U type;
00129     };
00130
00135     template <typename U>
00136     struct plain_to_reference {
00137       typedef const U& cref;
00138       typedef U& ref;
00139     };
00140     template <typename U>
00141     struct plain_to_reference<U&> {
00142       typedef U& cref;
00143       typedef U& ref;
00144     };
00145
00146   public:
00150     typedef un_const<T>::type non_const;
00155     typedef un_volatile<T>::type non_volatile;
00159     typedef un_volatile<un_const<T>::type>::type non_qualified;
00164     typedef plain_to_reference<T>::cref const_reference;
00169     typedef plain_to_reference<T>::ref reference;
00170   };
00171
00176   namespace {
00177
00181     struct nil {
00182     };
00183   }
00184
00190   template <typename H, typename T>
00191   struct typelist {
00195     typedef H head;
00199     typedef T tail;
00200   };
00201
00205   #define TYPELIST_1(T1) typelist<T1, nil>
00206
00209   #define TYPELIST_2(T1, T2) typelist<T1, TYPELIST_1(T2) >
00210
00213   #define TYPELIST_3(T1, T2, T3) typelist<T1, TYPELIST_2(T2, T3) >
00214
00217   #define TYPELIST_4(T1, T2, T3, T4) typelist<T1, TYPELIST_3(T2, T3, T4) >
00218
00224   template <typename T, typename list>
00225   struct index_of;
00226
00227   template <typename T>
00228   struct index_of<T, nil> {
00229     enum { position = -1 };
00230   };
00231
00232   template <typename T, typename tail>
00233   struct index_of<T, typelist<T, tail> > {
00234     enum { position = 0 } ;
00235   };
00236
```

```
00237   template <typename T, typename head, typename tail>
00238   struct index_of<T, typelist<head, tail> > {
00239   private:
00240     enum { temp = index_of<T, tail>::position };
00241   public:
00242     enum { position = (temp == -1) ? -1 : 1 + temp };
00243   };
00244
00248   template <typename T>
00249   class query {
00250   private:
00251     template <typename U>
00252     struct pointer_traits {
00253       enum { result = false };
00254     };
00255     template <typename U>
00256     struct pointer_traits<U*> {
00257       enum { result = true };
00258     };
00259
00260   public:
00266     typedef TYPELIST_4(
00267       unsigned char, unsigned short int,
00268       unsigned int, unsigned long int) unsigned_integrals;
00272     typedef TYPELIST_4(signed char, short int, int, long int) signed_integ
00276     typedef TYPELIST_3(bool, char, wchar_t) other_integrals;
00280     typedef TYPELIST_3(float, double, long double) floating_points;
00281
00282     enum { is_pointer = pointer_traits<T>::result };
00283     enum { is_unsigned_integral = index_of<T, unsigned_integrals>::positio
00284     enum { is_signed_integral = index_of<T, signed_integrals>::position >=
00285     enum { is_integral = is_unsigned_integral || is_signed_integral ||
00286       index_of<T, other_integrals>::position >= 0 };
00287     enum { is_floating_point = index_of<T, floating_points>::position >= 0
00288     enum { is_arithmetic = is_integral || is_floating_point };
00289     enum { is_fundamental = is_arithmetic || is_floating_point ||
00290            test<T, void>::is_equal_to };
00291   };
00292 }
00293
00294 #endif // __cphstl_type__
00295
00296
00297
```

---

copy.cpp File Reference

This file defines the function `copy`. More...

```
#include <iterator>
#include <type>
#include <cstring>
```

Go to the source code of this file.

Namespaces

namespace cphstl

Detailed Description

This file defines the function `copy`.

Original author

Jyrki Katajainen <jyrki@diku.dk>, December 2001

Sources

Andrei Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied*, Addison-Wesley (2001), see Section 2.10.5.

John Maddock and Steve Cleary, C++ type traits, *Dr. Dobb's Journal*, ***25***,*10 (2000), 38-44.*

Generated at Mon Dec 17 00:52:10 2001 for TheCopenhagenSTL by doxygen *1.2.3 written by Dimitri van Heesch, © 1997-2000*

copy.cpp

Go to the documentation of this file.

```
00001
00020
00021 #include <iterator> /* defines cphstl::iterator_traits */
00022 #include <type> /* defines cphstl::int2type and cphstl::query */
00023 #include <cstring> /* defines memcpy */
00024
00025 namespace cphstl {
00026
00027   namespace {
00028
00029     enum copy_algorithm_selector { conservative, fast };
00030
00033
00034     template <typename input_iterator, typename output_iterator>
00035     inline output_iterator
00036     copy (
00037       input_iterator p,
00038       input_iterator one_past_the_end,
00039       output_iterator r,
00040       cphstl::int2type<conservative>
00041     ) {
00042
00043       for (; p != one_past_the_end; ++p, ++r) {
00044         *r = *p;
00045       }
00046       return r;
00047     }
00048
00051
00052     template <typename input_iterator, typename output_iterator>
00053     inline output_iterator
00054     copy (
00055       input_iterator first,
00056       input_iterator one_past_the_end,
00057       output_iterator result,
00058       cphstl::int2type<fast>
00059     ) {
00060
00061       typedef std::iterator_traits<input_iterator>::difference_type size;
00062       size n = one_past_the_end - first;
00063       memcpy(result, first, n * sizeof(*first));
00064       return result + n;
00065     }
00066   }
00067
00144
00145   template <typename input_iterator, typename output_iterator>
00146   output_iterator
00147   copy (
00148     input_iterator first,
00149     input_iterator one_past_the_end,
00150     output_iterator result
00151   ) {
00152     typedef std::iterator_traits<input_iterator>::value_type input_element
00153     typedef std::iterator_traits<output_iterator>::value_type output_eleme
```

```
00154
00155     enum { algorithm_flag =
00156       cphstl::query<input_iterator>::is_pointer &&
00157       cphstl::query<output_iterator>::is_pointer &&
00158       cphstl::query<input_element>::is_fundamental &&
00159       cphstl::query<output_element>::is_fundamental &&
00160       sizeof(input_element) == sizeof(output_element) ? fast : conservativ
00161     };
00162     return copy(first, one_past_the_end, result, cphstl::int2type<algorithm
00163   }
00164 }
```
