GCC : From 2.95 to 3.2

Sébastien Ponce

Topics

- Simple changes
  - name of standard include files, std::endl, iostream, throw statements, vector iterators
- More complicated changes
  - string streams, parameterized macros , hash_map
- Finer C++ concepts
  - types of enumeration, typenames

Hors d'oeuvre (simple ones)

Include Files

No more .h at the end of standard include files

- 32 official include files, listed in the C++ standard in section 17.4.1.2
- compatible with gcc 2.95 No #ifdef

```
#include <math.h>
#include <stdio.h>
```

```
#include <math>
#include <stdio>
```

std::endl and co

Usage of Namespace “std::” is mandatory for objects of the standard library (endl, cout, …)

- compatible with gcc 2.95 No #ifdef

```
cout << “Hello World” << endl;
cerr << “Actually, there’s no errors” << endl;
```

```
std::cout << “Hello World” << std::endl;
std::cerr << “Actually, there’s no errors” << std::endl;
```

iostreams

cerr and cout are defined in <iostream>. You must include this file in order to use them.

Also don't forget the std namespace !

```
int void main (int argc, char** argv) {
   cout << "Hello World" << endl;
}
```

```
#include <iostream>
int void main (int argc, char** argv) {
   std::cout << "Hello World" << std::endl;
}
```

throw statements

If a class B inherits A and a virtual method A is redefined in B, the eventual throw specification in A must be there in B too.

- compatible with gcc 2.95 No #ifdef

```
class A {
  virtual void foo (int a) throws MyException;
}
class B {
  virtual void foo (int a);
}
```

```
class A {
  virtual void foo (int a) throws MyException;
}
class B {
  virtual void foo (int a) throws;
}
```

Iterators and pointers

An iterator on a vector type has nothing to do with a pointer to an element of the vector. In particular no cast between the two is allowed.

- compatible with gcc 2.95 No #ifdef

```
std::vector<int> myVec;
…
int* firstElement = myVec.begin();
```

```
std::vector<int> myVec;
…
std::vector<int>::iterator firstElement = myVec.begin();
```

Main dish
(More complex ones)

string streams

The standard library changed for string streams. The include file is now sstream (was strstream) and the types i/ostringstream (was i/ostrtream)

- **NOT compatible with gcc 2.95 use #ifdef**
- **compatible with windows same case for new compiler and windows**
- **uses now std::string as underlying type instead of char* code simplifications**

string stream example

```
const int buffer_size = 256;
char buffer[buffer_size] = {0, 0};
std::ostrstream ost(buffer, buffer_size);
ost << pv->physvolName << ":" << pv->tag();
const unsigned int len = strlen(ost.str());
char *resstr = new char[len+1];
strcpy(resstr, ost.str(), len);
resstr[len] = 0;
std::string result(resstr);
delete[] resstr;
```

(1)

```
#if defined (__GNUC__) && ( __GNUC__ <= 2 )
    (1)
#else
   std::ostringstream ost;
   ost << pv->physvolName << ":" << pv->tag();
   return ost.str();
#endif
```

parametrized macros

The usage of ## is more strict than before. One can no more put them if not needed

- compatible with gcc 2.95 No #ifdef

```
#define IMPLEMENT_SOLID(x) \
     static const SolidFactory<##x##> s_##x##Factory ; \
     const ISolidFactory&##x##Factory = s_##x##Factory ;
```

```
#define IMPLEMENT_SOLID(x) \
     static const SolidFactory<x> s_##x##Factory ; \
     const ISolidFactory& x##Factory = s_##x##Factory ;
```

hash_map

There is no hash_map in the standard library. Thus the include file hash_map of gcc 2.95.2 has become ext/hashmap. The type is no more std::hash_map but __gcc_cxx::hash_map

Solution : play with using keyword inside an #ifdef…

```
#ifdef _WIN32
  #include "stl_hash.h"
  using std::hash_map;
#else
  #if defined (__GNUC__) && ( __GNUC__ <= 2 )
    #include <hash_map>
    using std::hash_map;
  #else
    #include <ext/hash_map>
    using __gnu_cxx::hash_map;
  #endif
#endif
```

Dessert
(‘Tricky’ C++ Concepts)

Enum types

Enum and int types are no more related.

Thus you can no more use an int to initiate an object of type enum

```
enum smallInt { ZERO, ONE, TWO, THREE, FOUR };
smallInt a = 0;
```

```
enum smallInt { ZERO, ONE, TWO, THREE, FOUR };
smallInt a = ZERO;
```

Ambiguous templates

In some cases template declarations can be ambiguous.

```
int y;
template <class T> void g(T& v) {
   T::x(y);
   T::z * y;
}
```

- **T::x could be a function call (function x declared in class T) or a variable declaration (variable y of type T::x with perverse parenthesis).**
- **T::z * could be a type or y could be multiplied by the constant T::z**

Typeparams

In order to resolve the ambiguity, one has to tell the compiler whether T::x is a type

```
template <class T> void g(T& v) {
  typename T::x(y);
}
```

This is typically the case with containers

```
typedef std::vector<TYPE*> Data;
typedef Data::iterator (myIterator);
```

```
typedef std::vector<TYPE*> Data;
typedef typename Data::iterator (myIterator);
```

Why it's annoying

In many cases, the compiler can resolve the ambiguity by itself (and 2.95 did). But the C++ specification says :

- "The typename keyword is needed whenever a type name depends on a template parameter"

And gcc 3.2 sticks to it…

```
typedef std::vector<TYPE*> Data;
typedef Data::iterator myIterator;
```

```
typedef std::vector<TYPE*> Data;
typedef typename Data::iterator myIterator;
```

Conclusion

You don't have to change you code

- I'm doing it for you this week

But from now you must write gcc 3.2 compatible code

- In order to check it, compile it under gcc 3.2 on lxplus. Tou can change compiler with these two commands :

```
source $LHCBHOME/scripts/CMT.csh gcc32
source $LHCBHOME/scripts/CMT.csh gcc295
```