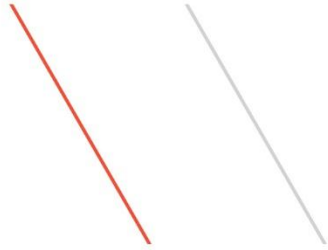


IBM z Systems

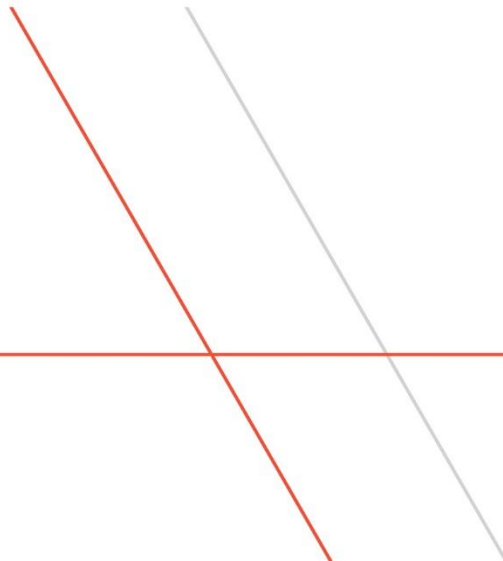


TPFUG – C++11 Support

Jim Johnston, TPF Development Lab

March 24, 2015

TPFUG – Dallas, TX



Presentation Summary

- Brief History of C++11 Standard
- z/TPF Compiler Support
- What C++11 Addresses
- High Value C++11 Features
- Other C++11 features worth noting
- Examples

C++11 History

- First two ISO standards were C++98 followed by C++03
 - TR1 (Technical Report 1)
 - proposal of extensions to C++03, eventually rolled into C++11
- First C++11 draft created in September of 2008
- C++11 approved by ISO in August of 2011 (ISO/IEC 14882:2011)
- GCC started implementing features for C++11 beginning with GCC 4.3 under the C++0x "experimental" moniker
 - Each successive release implemented more features of the C++11 standard
 - GCC 4.8 implements all of the major features of the C++11 standard
 - Implementation status can be found here:
<https://gcc.gnu.org/onlinedocs/libstdc++/manual/status.html#status.iso.2011>
- C++14 is the latest C++ standard ratified (ISO/IEC 14882:2014)

z/TPF Compiler Support - GCC

- GCC 4.1 & GCC 4.6
- GCC 4.6 compiler support was previously introduced in PUT 9 without any C++11 standard features
- PUT 11 provided support for C++11
 - Apars PJ42286 & PJ42291
 - Co-req'd GCC 4.6 tpf-11r1-11
- C++11 not turned on by default and is optional
 - `-std=c++0x` (compiler flag to enable the C++11 standard features)
 - `-std=gnu++0x` (compiler flag to enable GNU extensions)
- The lab may release support in the future which exploits C++11
 - Recommend using C++11 when possible.
- GCC 4.1 objects can co-exist with GCC 4.6/C++11 enabled objects


z/TPF Compiler Support – Systems/C++

- Systems/C++ V1.98 Compiler
 - APAR PJ42285
- V1.98 was the first Dignus Compiler version to introduce any C++11 support
- C++11 optional
 - -fcpp11 (compiler flag to enable the C++11 standard features)
- V1.98 now uses the libstdc++46 version of the C++ Runtime
 - Previously, Systems/C++ V1.96 shared the libstdc++41 runtime with GCC 4.1

What C++11 Addresses

- Some of the directives behind the drafting of C++11 (per Bjarne Stroustrup & ISO)
 - Maintain stability and compatibility with C++98/C++03
 - Evolve programming technique
 - Increase type safety by providing safe alternatives to earlier unsafe techniques
 - Increase performance and the ability to work directly with hardware
 - Implement "zero-overhead" principle (additional support required by some utilities must be used only if the utility is used)

High Value Items (lab's perspective)

- Pick up any new performance optimizations (from GCC & C++11)
- Concurrency
 - Before C11 standard C++ was required to use architecture dependent synchronizations or third party extensions for threads.
- Smart pointers
- Container classes
 - array (fixed-sized container)
 - forward_list (a singly-linked list)
 - unordered containers (hash tables)
 - Tuple (similar to std::pair)
- move constructor & move assignment
- delegated constructors 

Example:

```
class A{
public:
    A(): A(0){}
    A(int i): A(i, 0){}
    A(int i, int j) {
        num1=i;
        num2=j;
        average=(num1+num2)/2;
    }
private:
    ...
```

C++11 Miscellaneous Features worth noting

- regex
- constexpr
- strongly-typed nullptr & strongly typed enums
- iota
- set theory
 - all_of
 - any_of
 - none_of
- auto specifier

Strongly Typed Enums¹ (Before):

// fails at compilation!

```
enum Color {RED, GREEN, BLUE};  
enum Feelings {EXCITED, MOODY, BLUE};
```

Strongly Typed Enums (After)

```
enum class Color {RED, GREEN, BLUE};  
enum class Feelings {EXCITED, MOODY, BLUE};  
//accessing  
Color color = Color::BLUE;
```

auto example:

```
std::vector <string,string> myVec;
```

Before:

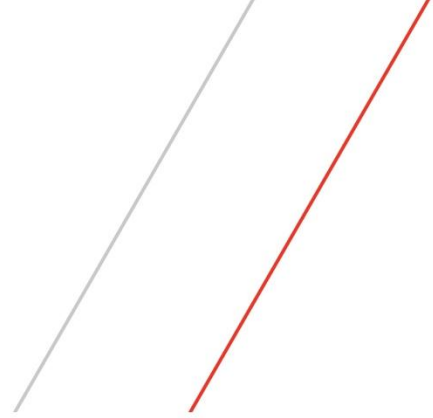
```
vector<string, string>::iterator itr = myVec.iterator();
```

After:

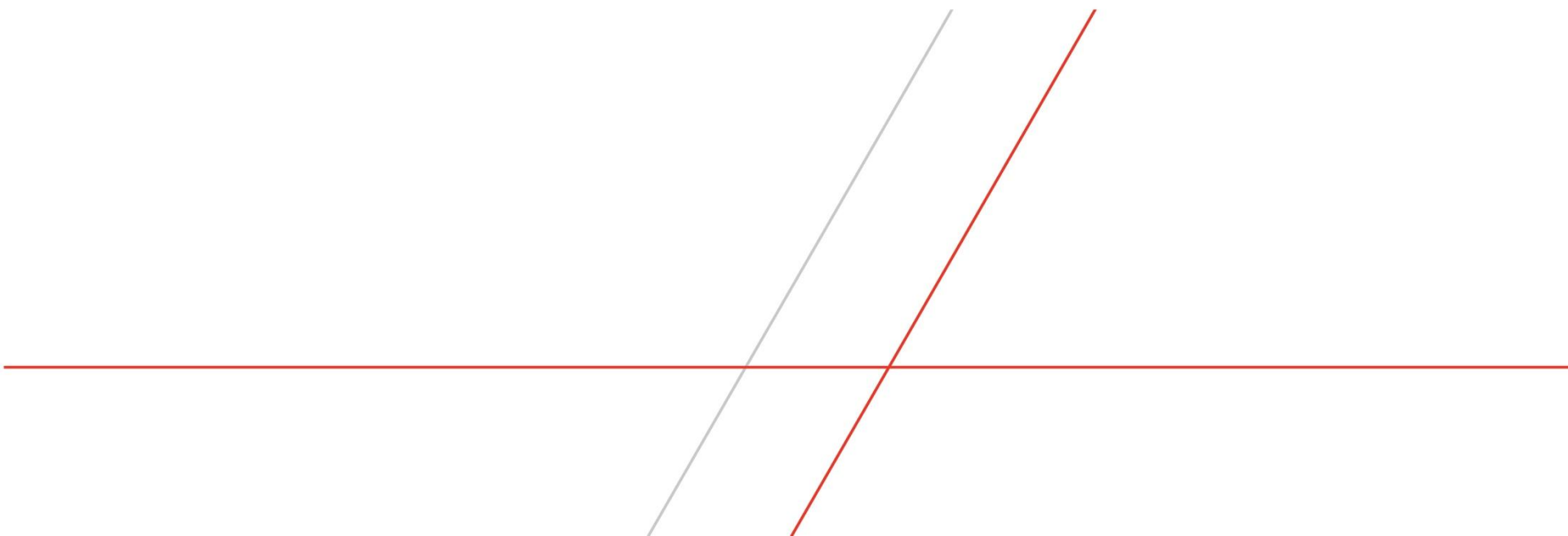
```
auto itr = myVec.iterator();
```

iota example:

```
vector<int> vInt(5);  
std::iota (vInt.begin(),vInt.end(),10);  
//10, 11, 12, 13,14
```

Examples



Concurrency – Starting a thread with parameters

Before C++11

Need to pull parms out

```
#include <pthread.h>

void myThreadFunction(void *arg) {
    struct ThreadParms *parms = (struct InParms *)arg;
    cout << parms->myint << "," << parms->mystr << endl;
}
```

Need to pass around Thread IDs

```
int main() {
    struct ThreadParms *parms;
    pthread_t thread_id;
    parms.myint = 3;
    parms.mystr = "message";
    retCode=pthread_create(&thread_id, NULL,
        &myThreadFunction, (void *) parms);
    retCode = pthread_join(thread_id, NULL);
}
```

After C++11²

```
#include <thread>

void myThreadFunction (int a, std::string &s) {
    cout << s << "and" << a << endl;
}

int main() {
    std::thread mythread (myThreadFunction, 3,
        std::string("message"));
    mythread.join();
}
```

Better experience passing parms to new thread.

Concurrency – Protecting data with lock_guard

Before C++11

```
#include <pthread.h>
pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;
std::list<int> mylist;
void updateListFunction(int value) {
    pthread_mutex_lock(mymutex);
    mylist.push_back(new_value);
    pthread_mutex_unlock(mymutex);
}
bool searchListFunction(int value) {
    pthread_mutex_lock(mymutex);
    found = std::find(mylist.begin(), mylist.end(), value)
    != mylist.end();
    ....
    if (error) {
        pthread_mutex_unlock(mymutex);
        throw error;
    }
    ....
    pthread_mutex_unlock(mymutex);
    return found;
}
```

Need to init mutex

What if an exception occurs!??

Need to make sure I unlock at all the possible error & return paths

After C++11³

```
#include <mutex>
std::mutex mymutex;
std::list<int> mylist;
void updateListFunction(int value) {
    std::lock_guard<std::mutex> guard(mymutex);
    mylist.push_back(new_value);
}
bool searchListFunction(int value) {
    std::lock_guard<std::mutex> guard(mymutex);
    found = std::find(mylist.begin(), mylist.end(),
    value) != mylist.end();
    ....
    if (error)
        throw error;
    ....
    return found;
}
```

One call to lock. Unlocks automatically.

Smart Pointers

unique_ptr example

```
std::unique_ptr<int> ptr1(new int(5));  
//copy constructor and assignment operators Deleted!  
std::unique_ptr<int> ptr2 = ptr1; //compilation fails  
std::unique_ptr<int> ptr2 = std::move(ptr1);  
ptr1.reset(); // will not delete object  
ptr2.reset();
```

shared_ptr example

```
std::shared_ptr<int> shptr1(new int(5));  
//managed refcount  
std::shared_ptr<int> shptr2 = shptr1;  
shptr1.reset(); // Does not delete int  
shptr2.reset(); // refcount = 0, deleted.
```

auto_ptr

Similar to unique_ptr but...
copy constructor and
assignment operator perform
moves.

weak_ptr example

```
std::shared_ptr<int> shptr1(new int(5));  
std::shared_ptr<int> wkptr = shptr1;  
std::shared_ptr<int> shptr2 = wkptr.lock();  
//ref count two integer  
shptr1.reset(); // int not deleted  
shptr2.reset(); // int deleted  
shptr1 = wkptr.lock(); // shptr1 == NULL
```

unordered_map, tuple, initializer list

```
#include <unordered_map>
#include <tuple>
using namespace std;
typedef tuple<int, string, string> myTuple_t;
typedef unordered_map<string, myTuple_t> myMap_t;

int main ()
{
    myMap_t MyMap;

    MyMap.insert({"key1", make_tuple (1, "Joe", "17 Palace St.")});
    MyMap.insert({"key2", make_tuple (2, "Greg", "14 Tuxedo Blvd.")});
    MyMap.insert({"key3", make_tuple (3, "Beth", "244 Laurel Rd.")});

    for (auto& x: MyMap)
        cout << get<0>(x.second) << get<1>(x.second) <<
get<2>(x.second);
    return 0;
}
```

Tuple Nice for aggregating
different types





Questions/Comments