Linux technical study
Minimum C++ program

by Eric Nicolas, enicolas@dvdfr.com
22 Jan 2005

Environement used:

Linux 2.6 (should work on any Linux 2.x)
Gcc 3.3 (probably required)
Intel x86 processor (required)

After having successfully built a minimum C program, the next question in my mind was: how about a C++ one? I knew already it was a much larger endeavour, but it proved feasible with a little help of the `gcc` toolchain.

In this article we start where we finished the minimum C program. We will just change the test.c sample working code to a C++ version using polymorphism, static variables and exceptions (in order to make sure non-trivial C++ mechanism are used and compiled in).

test.cpp

```
extern "C" void write(int file, const void *buffer, unsigned int length);
extern "C" int  strlen(const char *string);

void println(const char *string)
{
  write(1, string, strlen(string));
  write(1, "\n", 1);
}

class A {
public:
  A()                { println("A::A"); ++n; }
  virtual ~A()       { println("A::~A()"); }
  virtual int run() = 0;

  static int n;
};

class B : public A {
public:
  B()                { println("B::B"); }
  virtual ~B()       { println("B::~B()"); }
  virtual int run() { println("B::run()"); }
};

class C : public A {
public:
  C()                { println("C::C"); }
  virtual ~C()       { println("C::~C()"); }
  virtual int run() { println("C::run()"); }
};

int A::n = 0;

int main(int argc, char **argv, int envc, char **envp)
{
  A*a;

  a = new B();
  a->run();
  delete a;

  a = new C();
  a->run();
  delete a;

  try
  {
    B b;
    throw "exception";
  }
  catch(const char *message)
  {
    println(message);
  }

  return A::n;
}
```

This test program, if properly working, should display the following output and returns 3 (the number of instances created):

```
A::A
B::B
B::run()
B::~B()
A::~A()
A::A
C::C
C::run()
C::~C()
A::~A()
A::A
B::B
B::~B()
A::~A()
exception
```

Now if we try to link this program as we did with the C code, many symbols are missing. These are C++ internals required by Gcc:

1. new, delete operators

   ```
   operator new(unsigned)
   operator delete(void*)
   ```

2. exception handling helper functions

   ```
   __cxa_allocate_exception
   __cxa_throw
   __cxa_begin_catch
   __cxa_end_catch
   _Unwind_Resume
   __gxx_personality_v0
   ```

3. rtti helper classes

```
typeinfo for char const*
__cxa_pure_virtual
vtable for __cxxabiv1::__class_type_info
```

When you compile a C++ program using `g++` command, all those symbols are obtained from support libraries within the compiler. The exception handling low level helpers (_Unwind_xxx) are provided via the `libgcc_eh.a` private library. All gcc private headers, include and configuration files are located in a platform and version specific directory such as:

```
$GCC_BASE/lib/gcc-lib/i686-pc-linux-gnu/3.3.2/
```

The rest of the exception handling helpers, the operators and the rtti helper classes are defined in the `libsupc++`, a part of the gcc source tree, and provided in binary form within the C++ standard library `libstdc++`. Unfortunately we don't want to include this huge standard library in our minimal program, so we need a way to use the helper classes alone.

This proved easier than expected, as the `libsupc++` helper library is rather self contained. The trick is to create a Makefile which compiles this library alone, without the rest of the C++ standard library. The simplest is to have the makefile create symlinks to the required gcc source files and compile that as a standalone static library (see `/libc++/Makefile` in the source code).

Let's call this tiny helper library we just built `libmyc++.a`. Now our test program linked with `libgcc_en.a` and `libmyc++.a` only lacks a few C library standard calls:

```
void *malloc(unsigned int size);
void  free  (void *p);
void *memcpy(void *dst, const void *src, unsigned int len);
void *memset(void *dst, int c, unsigned int len);
void  abort ();
```

We simply provide C implementation for all of these required functions (see `/libc` directory in the source code). The memory allocators can be simply implemented as raw allocation from a pre-existing global bytes arena, although real-life programs would need a real allocator which would obtain memory from the kernel using the system call `sbrk`.

If we compile those dumb implementations (together with the `strlen` and `write` implementations already studied in the previous article) within a `libmyc.a` tiny library, our test program now sucessfully pass the link stage:

```
ld test.o -lgcc_en -lmyc++ -lmyc -ldl -o test
```

But at runtime it cannot start properly, a strange error happens when you launch the program:

```
# ./test
-bash: ./test: No such file or directory
```

This is because the program is not complete, it misses the startup C++ code required to initialize static variables such as A::n. It is the `collect2` gcc private tool which knows how to gather information about global constructors and generate this additional code. `collect2` is used instead of `ld` in the link stage:

```
$GCC_BASE/lib/gcc-lib/i686-pc-linux-gnu/3.3.2/collect2
--eh-frame-hdr -m elf_i386 -dynamic-linker /lib/ld-linux.so.2
test.o -lgcc_en -lmyc++ -lmyc -ldl -o test
```

After collecting all global constructors, it generates and compiles an additional assembly file which is dynamically loaded during the program startup (when linking with `ld`, the dynamic loader could not find this additional object at all, thus the odd 'No such file or directory' message).

Now the program works properly. The resulting binary is 35288 bytes long. That relatively big size is the required overhead for handling all C++ language mechanisms (rtti, exceptions, ...). Of course, myc, myc++ and gcc_en libraries can be provided as shared libraries in order to reduce the global overhead when several C++ programs are running.