C++ and the linker

Main

Quite some time ago I found a brillant and extensive article about linkers from Ian Lance Taylor. (see the appendix for a list of links to all 20 parts.) In part 13 he says: "There are some special challenges when using C++", and he is definitely right. I would like to elaborate on what theses challenges are and how the linker can be a serious trap for C++ developers. I learned it the hard way by spending hours and days on bug-hunting. Maybe this article will help others to avoid the frustration.

Basics

For those who don't know what a linker does, and who don't have time to read the 20 articles from Ian, I will give a brief introduction on the topic. I won't go into much detail, though.

How the Linker Works

Ian says that the demand for interoperability with FORTRAN and COBOL was the reason for introducing linkers. As far as I know there has been at least one other reason. As the compiler has to hold a lot of meta information about the code it compiles, the amount of available memory limits the maximum code size of a C program. Back in time, when C was invented, computer systems did not have much memory. To work around the resulting serious restriction of code size, the ANSI C standard (aka C89) states in section 5.1.1.1/1:

> A C program need not all be translated at the same time. The text of the program is kept in units called source files, (or preprocessing files) in this International Standard. A source file together with all the headers and source files included via the preprocessing directive #include is known as a preprocessing translation unit. After preprocessing, a preprocessing translation unit is called a translation unit. Previously translated translation units may be preserved individually or in libraries. The separate translation units of a program communicate by (for example) calls to functions whose identifiers have external linkage, manipulation of objects whose identifiers have external linkage, or manipulation of data files. Translation units may be separately translated and then later linked to produce an executable program.

The term "linkage" is defined in section 6.2.2/2:

> In the set of translation units and libraries that constitutes an entire program, each declaration of a particular identifier with external linkage denotes the same object or function. Within one translation unit, each declaration of an identifier with internal linkage denotes the same object or function. Each declaration of an identifier with no linkage denotes a unique entity.

So the linkage controls, which equally named identifiers denote the same entity. The linkage of an identifier is determined by a set of language rules. Long story short, the default is external linkage and can be overwritten to internal linkage by using the storage specifier `static`. Objects with automatic storage duration are an example of identifiers with no linkage.

Each identifier with external linkage must be defined exactly once in the entire program, as defined in section 6.9/5:

> [...] If an identifier declared with external linkage is used in an expression [...], somewhere in the entire program there shall be exactly one external definition for the identifier; [...]

This is the so-called "One Definition Rule". In this context "external definition" simply means that the definition is at global scope, i.e. outside of any function.

In other words the One Definition Rule demands that every identifier with external linkage is unique and originates from exactly one translation unit. For other translation units this identifier is foreign and can only by referenced.

In order to enable one translation unit to reference foreign identifiers, the identifiers must be declared within the unit. A declaration differs from a definition in such that it only introduces names to the translation unit without specifying how the object or function of that name looks like. It is sufficient to copy the required declarations into every translation unit manually. However, it is common practice to put those declarations into header files and include them in a translation unit whenever needed.

The ANSI C standard does not define how to merge multiple translation units into one executable program. This is entirely up to the toolchain. However, using a linker is suggested, and to my knowledge, all common implementations use a similar approach. In this article I will refer to the GNU Compiler Collection (GCC) version 4.3.2 on the Intel x86 Linux platform.

The GCC turns a translation unit defined by the source file `foo.c` into a so-called "object file" `foo.o`. An object file consists of various parts, only three of which are of interest in the scope of this article. These are:

- machine code: the binary instructions for the CPU, with placeholders for addresses of referenced identifiers.
- symbol table: list of
    - the names, the file-relative addresses, and the linkage type of all identifiers defined in the corresponding translation unit,
    - the names of all identifiers referenced in the corresponding tranlation unit, and
    - a couple more entries, which are not relevant here.
- relocation table: list of addresses of all placeholders in the machine code, together with the name of the referenced identifier

To be precise the linker does not handle identifiers, but symbols. But, as every definition of an identifier with linkage is represented by a symbol, the above explanations will suffice.

The linker is supplied with all object files, so it can read all symbol tables and do the relocations, thus substituting all placeholders with addresses. Finally the machine code of all object files is merged into one program by translating all addresses into one common address space.

Of course, in reality the whole process is more complex, but for the topic of this article these are the relevant steps.

Know Your Tools

The binutils are a set of tools for dealing with object files and libraries. One of those tools is `objdump`, which displays information from object files. So, how is it used?

First we write a header file, which will enable interoperability between two translation units by declaring a function `use_me`:

```
$ cat use_me.h
void use_me(int);
```

Next we write a C module which includes the header file and defines a function `bar` which calls `use_me`:

```
$ cat example.c
#include "use_me.h"

void bar()
{
        use_me(23);
}
```

Now we tell the compiler to process this translation unit. The -c switch prevents the compiler from linking a program, which it normally would do:

```
$ gcc -c example.c
```

Now let's see if we can find the above mentioned three parts. First we dump the code, retranslated from machine code to assembler code for human readability:

```
$ objdump --disassemble example.o
[...]
00000000 <bar>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   83 ec 08                sub    $0x8,%esp
   6:   c7 04 24 17 00 00 00    movl   $0x17,(%esp)
   d:   e8 fc ff ff ff          call   e <bar+0xe>
  12:   c9                      leave
  13:   c3                      ret
```

All of the above numbers are in hexadecimal notation. The first column shows the address of each machine instruction. The second one shows the bytes of the instructions while the third and fourth column show the corresponding mnemonics.

The first three commands create a stack frame for the function call to `bar`. The command at address 0x06 pushes the the parameter for the call to `use_me` on the stack. Finally `use_me` is callled and the stack frame of `bar` is destroyed.

The parameters of `call` may look weird, but remember that this is only a placeholder. The linker will replace it with the actual address of `use_me` later on. Let's check the relocation entry responsible for that:

```
$ objdump --reloc example.o
[...]
RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
0000000e R_386_PC32        use_me
```

This relocation entry instructs the linker to insert the address of the symbol `use_me` at address 0x0e. Among other things, the relocation type tells the linker the size of the address, which in this case is 32 bits.

Last let's take a look at the symbol table:

```
$ objdump --syms example.o
[...]
00000000 g     F .text  00000014 bar
[...]
```

Here we can see that the symbol `bar` resides in the object file `example.o` at address 0x0 and has a size of 0x14 bytes. This information could be used to do relocations, which are needed if `bar` is referenced by some other translation unit.

If you have to find out why the linker reports "unresolved symbol" or "multiple definition" errors, `nm` is your tool of choice:

```
nm example.o
00000000 T bar
         U use_me
```

The output confirms that the symbol `bar` resides at the address 0x0. `T` stands for "text section" which is the code part of the object file. Furthermore the output says that the symbol `use_me` is used but not defined. `U` here stands for "unresolved".

Hereditary Disease

The ISO C++ language inherits from C90, which is the ISO variant of ANSI C. Therefore there are a lot of similarities between ANSI C and C++. Obviously most vendors decided to simply adapt their existing C toolchains to support C++ instead of creating a new one.

Unfortunately C++ has several language features, which can't be handled well by a toolchain designed with ANSI C principles in mind. Therefore the C++ support of various toolchains is more like a hack than a clean design. In combination with various gotchas in the language itself this is a constant source of headaches for developers. From the next section on I will illustrate this with some examples.

To make things worse common C linkers already had major flaws:

```
$ cat foo.c
struct Broken {
    char c;
    int i;
};

struct Broken broken;

void init()
{
    broken.i = 42;
}

$ cat main.c
#include <stdio.h>

struct Broken {
    int i;
};

struct Broken broken;
void init();

int main()
{
    init();
    printf("%d\n", broken.i);
    return 0;
}

$ gcc -Wall -o main main.c foo.c

$ ./main
0
```

The above program is ill-formed, because the differing definitions of struct `Broken` violate the One Definition Rule. So it doesn't take wonder that the result of the above program is not 42 as one would expect. But, concerning the GCC toolchain, neither the compiler nor the linker detects the error in the program. Which is by design, because neither of them can see it. When the compiler processes one translation unit it does not even know that others exist. And the linker has no access to the type definitions. It isn't even aware of the existence of different types.

With ANSI C this behaviour is fine, because it is not specified, who is responsible to enforce the One Definition Rule. C++ is an improvement to C90 in so far that more details are specified. However, at this point, it just adds that "no diagnostic [is] required" (section 3.2/3), which means according to section 1.4/2:

> If a program contains a violation of a rule for which no diagnostic is required, this International Standard places no requirement on implementations with respect to that program.

Great!

Still it is important to note, that the linker could enforce the One Defintion Rule with support from the compiler and the object file format. For every definition the compiler could augment the object file with additional information that can be used to test for equality. It is the fault of the toolchain that this is not done. And it is the fault of the language that the checks are optional. This is true for both C and C++.

Name Mangling

Let's see how the C++ toolchain works:

```
$ cat use_me.h
void use_me(int);

$ cat example.cc
#include "use_me.h"

namespace foo {
    void bar()
    {
        use_me(23);
    }
}

$ g++ -c example.cc

$ nm example.o
         U _Z3use_mei
00000000 T _ZN3foo3barEv
         U __gxx_personality_v0
```

This looks strange. First, there is an unresolved symbol named "__gxx_personality_v0" for which there is no identifier in our code. This comes from the fact that the C++ compiler implicitly injects code into the translation units to enable C++ features like dynamic initialization and exception handling.

Second, the names of the symbols are decorated with additional strings. This is called name mangling.

The decoration before the identifier name is needed because C++ supports namespaces. For example the same function name can occur multiple times in different namespaces while denoting a different entity each time. To enable the linker to differentiate between those entities the name of each identifier is prepended with tokens representing its enclosing namespaces.

The decoration after the identifier name is needed because C++ allows function overloading. Again the same function name can denote different identifiers, which differ only in their parameter list. To enable the linker to differentiate between those, tokens representing the parameter list are appended to the name of the identifier. The return type of a function is disregarded, because two overloaded functions must not differ only in their return type.

Current GCC version use the specifications of the Itanium C++ ABI for name mangling. The encoding is straight forward but still cumbersome the decode manually. Fortunately `nm` provides the `-C` switch which demangles the symbol names to provide human readability:

```
$ nm -C example.o
         U use_me(int)
00000000 T foo::bar()
         U __gxx_personality_v0
```

In my opinion name mangling is a hack in order to be able to stick with the old C linker. And the ISO C++ standard does not define how it should be done as it doesn't even require a linker at all, just like the ANSI C standard. So each toolchain implementation uses its own encoding. If the toolchain or the object file format limits the size of symbol names, this can limit the maximum nesting depth of namespaces drastically.

Furthermore it is impossible to write platform-independent C++ code when using dynamic library loading with dlopen. This is, because the symbol name, which is required by `dlsym` in order to retrieve the address of a symbol, is platform dependent. The typical solution for this problem is to use some interfacing function which is declared with `extern "C"`. The meaning of this so-called "language linkage" is implementation defined. With GCC the C language linkage enables the usage of the C application binary interface (ABI), which in turn disables some C++ features like function overloading.

```
$ cat use_me.h
extern "C" void use_me(int);
```

```
$ g++ -c example.cc

$ nm example.o
00000000 T _ZN3foo3barEv
         U __gxx_personality_v0
         U use_me
```

The reason why C++ does not work well with `dlopen` is that `dlopen` is a POSIX function, and POSIX and ISO C++ known nothing about each other. Thus, it takes no wonder that problems show up if both are used in combination.

Dynamic Initialization

One of the design goals of C++ was to make all user defined types first-class objects [1]. Consequentially an object with static storage may be of an arbitrary type. C only supports so called Plain Old Data (POD) objects, which can be initialized by constant expressions. C++ also knowns non-POD objects, and they must be initialized by a call to their constructor instead. This is called dynamic initialization, in contrast to static initialization, which is done otherwise.

In C, "all objects with static storage duration shall be initialized (set to their initial values) before program startup." (ANSI C, section 5.1.2/1). In C++ things are a bit different, though. Section 3.6.2/3 states:

> It is implementation-defined whether or not the dynamic initialization [...] of an object [...] is done before the first statement of main. If the initialization is deferred to some point in time after the first statement of main, it shall occur before the first use of any function or object defined in the same translation unit as the object to be initialized.

If the constructor accesses other objects with static storage we have dependencies between them, i.e. they have to be initialized in the right order to avoid accessing uninitialized objects. So who will guarantee this? Well, the programmer, of course. The compiler can't do it, because the correct initialization sequence depends on user defined code, which may be arbitrarily complex. Finding the right initialization sequence in every case is related to the Halting Problem which is undecidable. In fact, the compiler simply does not care about this problem:

```
$ cat example.cc
#include <iostream>

class A {
public:
    A();
};

class B {
public:
    B() { std::cout << "B::B" << std::endl; }
    void access() { std::cout << "B::access" << std::endl; }
};

static A a;
static B b;

A::A()
{
    std::cout << "A::A" << std::endl;
    b.access();
}

int main()
{
    std::cout << "main" << std::endl;
    return 0;
}

$ g++ -o example example.cc
$ ./example
A::A
B::access
B::B
main
```

We see that `b` is accessed before it is constructed. This is not good and hard to find as a bug. As long as `access` doesn't dereference any invalid pointers, this bug will not cause a segmentation fault, as the memory is static memory and thus valid. Instead, during `access` members of `b` contain arbitrary values, and write accesses to them will be overwritten by the constructor later.

So what can the programmer do to avoid the problem? The answer is quite simple. First, the standard guarantees that static initialization is done before any dynamic initialization. Second, the standard states that dynamic initialization is done in the same order in which the definitions appear in the translation unit. So the above example could be fixed by switching the two lines which define the objects `a` and `b`.

This works for single translation units. But what if there are dependencies of objects with static storage across translation units? What is the order of appearance then? The answer is simple. It is implementation defined.

Some toolchains take the order, in which the object files are passed to the linker, as the primary sort criterion. Others maybe not. So if your program works for you, it still may show subtle bugs due to initialization order problems on a different platform. This is called the static initialization order fiasco.

The solution to this problem is a hack which uses another kind of static objects, namely static local objects. Since objects of this kind are initialized as soon as their definition is passed initially at run time, they can force an initialization order. To demonstrate how this technique works for dependencies between translation units, I moved `B` to a different source file and added a header file of its own:

```
$ cat b.h
class B {
public:
    B();
    void access();
};

B& getB();

$ cat b.cc
#include <iostream>
#include "b.h"

B& getB()
{
    static B b;
    return b;
}

B::B()
{
    std::cout << "B::B" << std::endl;
}

void B::access()
{
    std::cout << "B::access" << std::endl;
}

$ cat a.cc
#include <iostream>
#include "b.h"

class A {
public:
    A()
    {
        std::cout << "A::A" << std::endl;
        getB().access();
    }
};

static A a;

int main() { return 0; }

$ g++ -o example a.cc b.cc

$ ./example
A::A
B::B
B::access
```

The global object of type `B` is accessed only through the getter function `getB`. The moment this function is called the first time, the object is initialized right away.

Linker Dependencies

One of the tasks the GNU linker performs is to leave unneeded code out of the program. If you link against some huge library only the parts that you actually use in your program are included in the program, thus keeping it small. But what does "code is needed" actually mean?

In the C world the answer is easy: starting from the translation unit where `main` is defined every referenced symbol causes the object file it originates from to be included. The same rule applies recursively to all symbols referenced in the included object file. Apart from the fact that the object file granularity is a bit coarse this works fine.

For C++, things are not that easy, because dynamically initialized objects can execute arbitrary code. Here is a simple example for this:

```
$ cat setting.h
#include <string>

class Setting {
public:
    Setting(std::string key, std::string value);
    static std::string get(std::string name);
};

$ cat setting.cc
#include "setting.h"
#include <map>

typedef std::map<std::string, std::string> SettingMap;

SettingMap& map()
{
    static SettingMap map;
    return map;
}

Setting::Setting(std::string key, std::string value)
{
    map()[key] = value;
}

std::string Setting::get(std::string key)
{
    return map()[key];
}
```

With this basic implementation one can have a compile time configuration like this:

```
$ cat config.cc
#include "setting.h"
```

```
static Setting host("host", "localhost");
static Setting port("port", "423");
// ...
```

And access the configuration like this:

```
$ cat foo.cc
#include "foo.h"
#include "setting.h"
#include <iostream>

void init()
{
    std::string host = Setting::get("host");
    std::cout << "host is '" << host << "'" << std::endl;
}
```

Let's test the setup:

```
$ cat foo.h
void init();

$ cat main.cc
#include "foo.h"

int main()
{
    init();
    return 0;
}

$ g++ -o main main.cc foo.cc config.cc setting.cc

$ ./main
host is 'localhost'
```

So far, so good. But what if we ship this implemenation as a library?:

```
$ g++ -c foo.cc config.cc setting.cc

$ ar r libfoo.a foo.o config.o setting.o
ar: creating libfoo.a

$ g++ -o main main.cc libfoo.a

$ ./main
host is ''
```

Oops! The host is not set. What happened is that `config.o` is not used by `main` so it is not linked into the binary. The linker just doesn't know that its good old rules for dependency tracking are insufficient for C++. As a consequence the objects `host`and `port` do not exist in the program. Thus the side effects of the `Setting` constructor don't come into effect, leaving the settings map empty. In the end the query results in an empty string.

To understand why it worked first I have to expand on the above explanation. The linker drops unused object files only if they originate from a library. Object files explicitly passed to the linker are always linked into the program. So this works, too:

```
$ g++ -o main main.cc libfoo.a config.o

$ ./main
host is 'localhost'
```

Alternatively there are linker options to overwrite the default behavior and include all object files from an archive. But use them with care to keep your programs small.

Inline Functions

Inline functions are another C++ language feature. The idea behind those is that if a function executes very few operations, the costs for the function call are very high compared to what the function actually does. A typical example for this is the `intmax(int, int)` function, which returns the maximum of the two parameters.

The traditional C way is to define a preprocessor macro to make occurrences of `max` look like function calls, although the preprocessor actually does a text substitution. However, this has a lot of disadvantages, such as loss of type safety andmultiple execution of side effects of the parameters.

In C++ you can write a `max` function and tell the compiler that inlining is preferred. In case the compiler decides to actually do the inlining, it replaces the function call with code that semantically does the same as the function call would have.

Problems arise when an inline function is used across translation units. In order to do the substitution the compiler needs the definition of the function at every location where it is used. However, there is no simple way for the compiler to retrieve the definition from other translation units, because by design it doesn't know anything about them.

That is why section 7.1.2 of the ISO C++ standard says:

> An inline function shall be defined in every translation unit in which it is used and shall have exactly the same definition in every case (3.2).

The referenced section 3.2 specifies the One Definition Rule. By requiring the same definition in every translation unit, compliance to the One Definition Rule is somewhat artificially brought about. In order to follow section 7.1.2 definitions of inline functions are usually put into some header file.

Rules Without Enforcement Mean Nothing

As mentioned in the chapter Hereditary Disease, there is no instance that ensures the One Definition Rule, which is true for inline functions as well. So, what happens if the assumption "all definitions are the same" turns out to be false. Let's see:

```
$ cat foo.cc
#include <iostream>

inline void doSomething() { std::cout << "foo" << std::endl; }
void run_foo() { doSomething(); }

$ cat bar.cc
#include <iostream>

inline void doSomething() { std::cout << "bar" << std::endl; }
void run_bar() { doSomething(); }

$ cat main.cc
void run_foo();
void run_bar();

int main()
{
    run_foo();
    run_bar();
    return 0;
}

$ g++ -o example *.cc

$ ./example
bar
bar
```

This is not a typo. The output is twice "bar". This means that `run_foo()` calls `doSomething` from `bar.cc` instead of its own implementation.

Before showing you what happens behind the scenes let me emphasize that the above situation can easily occur. It is not a corner case. Here is what happened to a colleague of mine:

He needed multiple handlers for some UI framework, which he chose to implement in different source files each. Since the name of the handler class did not really matter he just had all handlers have the same name. In addition to that he defined the handler functions within their class definition to save typing.:

```
$ cat keyboard_handler.cc
#include <uihandler.h>

class Handler : public UIHandler {
public:
    void handle(Event*) { /* some implementation */ }
};
static Handler keyboard_handler;

$ cat mouse_handler.cc
#include <uihandler.h>

class Handler : public UIHandler {
public:
    void handle(Event*) { /* some different implementation */ }
};
static Handler mouse_handler;
```

One would expect that this would lead to a multiple definition error from the linker, because `Handler::handle` is defined twice. But it doesn't. The linker is fine with the above code and produces a broken program.

The reason for this behavior is section 7.1.2.3 of the ISO C++ standard, which states that "a function defined within a class definition is [implicitly] an inline function". Maybe this is because the above technique is mostly used for short functions such as getter and setter functions, which are worthwhile for being inlined.

Anyway, my colleague ended up in the exact same situation as described in the initial example . One handler implementation was disregarded and mouse events were handled by the keyboard handler instead.

Optimizing

Now let's see what happens behind the scenes when the One Definition Rule for inline functions is violated:

```
$ cat foo.cc
#include <iostream>

inline void doSomething() { std::cout << "foo" << std::endl; }
void run_foo() { doSomething(); }

$ g++ -c foo.cc

$ nm -C foo.o | grep doSomething
00000000 W doSomething()
```

We can see that the `doSomething` symbol has the special type `W`, which stands for "weak symbol". Weak symbols are subject to special linker rules, which prevent multiple definition errors, which is necessary to comply to the language rules for inline functions. So, if for some reason symbols for different identifiers should have the same name, all but one of those symbols are dropped silently.

In case you're wondering why there are symbols for `doSomething` in the first place, remember that the keyword `inline` is only an expression of preferrence. As a matter of fact, GCC refuses to inline functions when optimization options are disabled. So in the above example `doSomething` remains a normal function in the translation unit. Still, the compiler must apply the semantics of inline functions to it, i.e. multiple definition issues are disregarded. Thus `doSomething` becomes a weak symbol.

Let's crosscheck:

```
$ g++ -c -O1 foo.cc
$ nm -C foo.o | grep doSomething
```

As you can see there is no symbol for `doSomething` now. The function was actually inlined. Let's recheck the initial example, this time with optimization options enabled:

```
$ cat foo.cc
#include <iostream>

inline void doSomething() { std::cout << "foo" << std::endl; }
void run_foo() { doSomething(); }

$ cat bar.cc
#include <iostream>

inline void doSomething() { std::cout << "bar" << std::endl; }
void run_bar() { doSomething(); }

$ cat main.cc
void run_foo();
void run_bar();

int main()
{
    run_foo();
    run_bar();
    return 0;
}

$ g++ -o example -O1 *.cc

$ ./example
foo
bar
```

Now the `doSomething` functions are inlined without leaving any symbols the linker could mix up. So apparently everything works fine. But if you start debugging and thus turn optimization off the bug shows up. This is an inverted heisenbug. Very nasty beast it is.

One last thing: In some cases the compiler has to create a symbol although it inlines the function. This can happen for example when a function pointer references the function. So, the above phenomenon does not always disappear when optimization is enabled.

Unnamed Namespaces

If you want to be safe against accidental overwriting of symbols you must avoid that names of different identifiers clash with each other. A practical way to do so is using unnamed namespaces. Section 7.3.1.1/1 defines them as follows:

> An unnamed-namespace-definition behaves as if it were replaced by:
>
> ```
> namespace unique { /* empty body */ }
> using namespace unique;
> namespace unique { namespace-body }
> ```
>
> where all occurrences of `unique` in a translation unit are replaced by the same identifier and this identifier differs from all other identifiers in the entire program.

The uniqueness of the assigned names ensures that an identifier in such an unnamed namespace does not conflict with identifiers from other translation units. Still unnamed namespaces are transparent within their translation unit.

Let's see what the GCC does:

```
$ cat foo.cc
```

```
namespace {
    void hidden() { }
}

$ g++ -c foo.cc

$ nm foo.o
00000000 t _ZN12_GLOBAL__N_13hiddenEv

$ cat bar.cc
namespace {
    void hidden() { }
}

$ g++ -c bar.cc

$ nm bar.o
00000000 t _ZN12_GLOBAL__N_13hiddenEv

$ nm -C foo.o bar.o
foo.o:
00000000 t (anonymous namespace)::hidden()

bar.o:
00000000 t (anonymous namespace)::hidden()
```

The GCC is cheating here. Instead of assigning unique names to the unnamed namespaces, it assignes the same name to all of them. In order to effectively follow the language rules, it creates the symbols from those namespaces with local linkage, as indicated by the lower case `t` above. The reason for this hack is comprehensible. How could the compiler find a program-wide unique name without having a program-wide overview when processing a single translation unit?

Templates

Templates have been introduced rather late in the evolution of C++ [1]. Thereby their intregration did not fit smoothly into the language and the existing toolchains. This section shows some issues raising from this fact.

This won't be a complete documentation on templates. This article would become a book otherwise, and there already are plenty of books on this topic. So this will be a brief introduction only.

Templates are code generators. First you tell the compiler how something looks in general, by defining parametrized code. This is called a template definition. Among other things a parameter can be an arbitrary type. When you actually use the template you assign particular values to the parameters. When processing the template instantiation the compiler takes the template definition, substitutes the placeholders for the parameters with the provided values and compiles the resulting code.

Assume you need a stack implementation for two different types `A` and `B`. In this case it would be desirable to be able to provide a generic implementation, which will work for both types. One possible way to achieve this is to have `A` and `B`implement a common interface `contained`, and have the container handle references to objects of the type `contained`. This technique is called polymorphy. This approach has two major disadvantages, though. First, you need to manually type-cast the returned object references. Second, type checks for the casts are done at run time, although the involved types are already known at compile time. This is a waste of run time resources.

For the above example a template stack implemenation, which can be instantiated by both `A` and `B`, would provida a way to have the compiler do the casts and type checks at compile time. Such a stack implementation could look like this:

```
$ cat stack.hxx
#include <assert.h>
template<typename T>
class Stack {
public:
    Stack(int size) :
        data(new T[size]),
        max_size(size),
        current_size(0)
    { }

    ~Stack() { delete [] data; }

    void push(T element)
    {
        assert(current_size < max_size);
        data[current_size++] = element;
    }

    T pop()
    {
        assert(current_size > 0);
        return data[current_size--];
    }
private:
    T* data;
    const int max_size;
    int current_size;
};
```

This is the template definition. The template parameter is `T`, which represents an arbitrary type. More is not known at this point.

```
Stack<A> myStack(23);
Stack<B> myStack(42);
```

This is the template instantiation. At this point the compiler generates the classes `Stack<A>` and `Stack<B>`, and handles them just like any other class from there on.

Aside from class templates, C++ also supports function templates, which differ from the former. However, for the scope of this article these differences are irrelevant.

A Lame Excuse

The above stack implementation example is already rather advanced. Someone new to C++ would probably implement something that, besides template specific additions, would look like a normal class declaration and definition:

```
$ cat stack.hh
template<typename T>
class Stack {
public:
    Stack(int size);
    ~Stack();
    void push(T element);
    T pop();

private:
    T* data;
    const int max_size;
    int current_size;
};

$ cat stack.cc
#include "stack.hh"
#include <assert.h>

template<typename T> Stack<T>::Stack(int size) :
        data(new T[size]),
        max_size(size),
        current_size(0)
    { }

template<typename T> Stack<T>::~Stack() { delete [] data; }

template<typename T> void Stack<T>::push(T element)
{
    assert(current_size < max_size);
    data[current_size++] = element;
}

template<typename T> T Stack<T>::pop()
{
    assert(current_size > 0);
    return data[current_size--];
}
```

So let's try to use it:

```
$ cat main.cc
#include "stack.hh"

int main()
{
    Stack<int> s(23);
    s.push(1);
}

$ g++ -c stack.cc

$ g++ -o main main.cc stack.o
main.cc:(.text+0x22): undefined reference to `Stack<int>::Stack(int)'
main.cc:(.text+0x35): undefined reference to `Stack<int>::push(int)'
main.cc:(.text+0x40): undefined reference to `Stack<int>::~Stack()'
main.cc:(.text+0x63): undefined reference to `Stack<int>::~Stack()'
```

What is wrong? Why is the linker unable to find the implementation in `stack.o`? Let's find out:

```
$ nm -C stack.o
```

Apparently `stack.o` has no symbols. Why is that?

The answer is again: "different translation units". When the compiler processes `stack.cc` it can't know that `Stack` with `T=int`will be needed, because it is not aware of `main.cc`. And when the compiler processes `main.cc` it doesn't see the implementation of `Stack` in order to do the instantiation. So the compiler can't do anything but list the unresolved symbols in`main.o` in the hope that the instantiation is done in some other translation unit.

The standard defines the `export` keyword for such a case. Section 14/8 states:

> [...] An exported template need only be declared (and not necessarily defined) in a translation unit in which it is instantiated.

Obviously the common workflow of C++ toolchains, as described above, can not handle what the export feature requires. TheComeau C++ toolchain supports export by delaying the instantiation up to the point when prelinking is done. This means that the compiler leaves a note for the linker, which calls the compiler back in order to do the instantiation. To my knowledge, every other C++ toolchain is not able to perform such callbacks. Instead they just ignore the `export` keyword:

```
$ cat example.cc
export template <class T> T const& min(T const&, T const&);

int main()
{
    return min(2,3);
}

$ g++ -c  example.cc
example.cc:1: warning: keyword 'export' not implemented, and will be ignored
```

Formally, the C++ standard is fine, as `export` provides a clean solution for templates. But in fact you have to fiddle around to get along with them. This is particularly true when you want to write portable code.

So how to get templates to work? The only translation unit where the template implementation is available is `stack.cc`. So we add an explicit template instantiation there:

```
$ cat stack.cc
[...]
template class Stack<int>;

$ g++ -c stack.cc

$ nm -C stack.o
00000000 W Stack<int>::pop()
00000000 W Stack<int>::push(int)
00000000 W Stack<int>::Stack(int)
00000000 W Stack<int>::~Stack()
[...]

$ g++ -o main main.cc stack.o
```

This works. But pinning the values for `T` breaks our initial goal of having a generic implementation. There must be a better solution.

Déjà-vu

In order to make generic template implementations function properly, you have to use an implementation like in the exampleof `stack.hxx`: Definition and implementation go together in a header file, which is included wherever needed, so that the compiler can do the instantiations. Let's check:

```
$ cat main.cc
#include "stack.hxx"

int main()
{
    Stack<int> s(23);
    s.push(1);
}

$ g++ -c main.cc

$ nm -C main.o
00000000 W Stack<int>::push(int)
00000000 W Stack<int>::Stack(int)
00000000 W Stack<int>::~Stack()
[...]

$ g++ -o main main.o
```

Works fine. The required symbols are all in `main.o`, because this is where the instantiation was done.

But wait, the symbols are marked as "weak symbols". Sure enough the linker should not reject a program with multiple definition errors, if for some reason two translation units contain the same instantiation. Therefore weak symbols are the way to go.

But beware: as we already have seen the linker discards all but one weak symbol of the same name, without checking if they all represent the same identifier. The chapter Inline Functions showed how much of a trap this is. And actually we face the exact same problems here. This is what happened to me once:

```
$ cat types.hh
enum Type {
    A,
    B
};

template<Type SELECT> struct Traits;

template<> struct Traits<A> {
    static const int value=23;
};

template<> struct Traits<B> {
    static const int value=42;
};
```

As you can see enum types can be template parameters as well. On top of that the above code defines two template specializations. They tell the compiler to use a different implementation if the template parameters have some specific value. This is used by a technique called traits. It provides a compile time mapping, which in this case is between enum and integer constants. It is used like this:

```
$ cat foo.cc
#include <iostream>
#include "types.hh"

template<Type SELECT>
void map()
{
    std::cout << "foo " << Traits<SELECT>::value << std::endl;
}

void setup_foo()
{
    map<A>();
    map<B>();
}

$ cat foo.hh
void setup_foo();

$ cat main.cc
#include "foo.hh"

int main()
{
    setup_foo();
}

$ g++ -o main main.cc foo.cc

$ ./main
foo 23
foo 42
```

In addition to `foo` I had another module `bar`, which had to do a similar job:

```
$ cat bar.cc
#include <iostream>
#include "types.hh"

template<Type SELECT>
void map()
{
    std::cout << "bar " << Traits<SELECT>::value << std::endl;
}

void setup_bar()
{
    map<A>();
    map<B>();
}

$ cat bar.hh
void setup_bar();
```

If we adapt main to call into both modules, we trigger the bug:

```
$ cat main.cc
#include "foo.hh"
#include "bar.hh"

int main()
{
    setup_foo();
    setup_bar();
}

$ g++ -o main main.cc foo.cc bar.cc

$ ./main
foo 23
foo 42
foo 23
foo 42
```

There it is. `setup_bar` is calling `map` from `foo.cc` instead of its own implementation. Everything is analogous to the explanations from the chapter Inline Functions. And again this is not a corner case but happens to C++ developers in the real world.

Conclusion

C++ is the language of pragmatism. It works and it's efficient. Starting from the first pragmatic decision, namely having "C with classes", that's all ever counted. Whenever I wondered: "WTF? Why is this?", I found good reasons for every single language decision, with efficiency and compatibility being the most frequent ones.

In my opinion Bjarne Stroustrup et al. did a good job in meeting their design goals. Yet, I still believe that C++ is a dead end. The C heritage is a heavy burden. This article has lined out a mere few examples for this and on my blog there are some others (1, 2, 3, 4, 5). The fact that Bjarne Stroustrup et al. uncompromisingly pursued the design goals of efficiency and compatibility resulted in a language, which is very difficult to understand and use (6, 7, 8): Hundreds of special rules for special cases (9, 10, 11, 12, 13), language features that clash when used in particular combinations (14, 15), undefined and implementation-defined behavior everywhere (16, 17). Nevertheless the foundations of C++ have been remaining archaic and defective. C++ 0x will not change this. Instead it is going to add even more features, more rules and more complexity. In my opinion this is the wrong way to go.

If you are wondering why I still keep myself busy with C++, take a look at The Computer Language Benchmarks Game. Sad but true, when efficiency matters, there is no alternative to C or C++. At logix-tt I have implemented a cryptographic communication service in C++, which was designed for thousands of simultaneous client connections. The customer-facing clients were implemented with Java Server Pages (JSP). While the server running the communication service was idle most of the time, we needed more and more server machines to host the tomcat instances for the JSPs. How come?

It appears to be commonly accepted, that comfortable-to-write and efficient-to-run are mutually exclusive. However, I have never seen proof of this, and I beg to differ. High level programming languages are so inefficient, because efficiency has never been a design goal for them. I think it's time for a new approach, which provides a convenient way of writing efficient implementations.

In my opinion, pattern-based programming looks like a promising approach. The idea behind it is that a program is a collection of instances of patterns. For each of these patterns the compiler knows decidable algorithms for their translation into efficient target code. The key point is that the compiler gets to know more about the semantics of the code. This additional information often turns previously undecidable problems into decidable ones. This way as many decisions as possible can be made at compile time instead of having to waste runtime resources on them.

Under the lead of Volker Birk we at logix-tt are researching the viability of such an approach. Hopefully there will be some publications on this topic from me and other developers from logix-tt soon.

Appendix

Thanks to

- Ansgar Wiechers
- Lars Rohwedder
- Andreas Bernauer
- Liesa Keizer

Links to Ian's Linker Articles

- part 1
- part 2
- part 3
- part 4
- part 5
- part 6
- part 7
- part 8
- part 9
- part 10
- part 11
- part 12
- part 13
- part 14
- part 15
- part 16
- part 17
- part 18
- part 19
- part 20