Slide 1.

Hi!

First of all I would like to thank you for your coming!

It’s nice to see that you came here in your free time to listen about C++ linker! I really appreciate that you’ve made your choice in that way.

So, today we will take a look at C++ linker more precisely. We use this magic tool every day, but it’s work is mostly hidden from us. We only remind ourselves about C++ linker when we see strange errors like “unresolved symbols … smth smth”. But still it will be good to know about possible traps that we can get in, right?

Even though we will cover some basic things about work of linker, I’m sure that there will be a lot of interesting information for advanced C++ developers.

I prepared quite a big presentation for today so we will split into parts with a short break when you can ask some questions or go for a cup of coffee.

Please be ready for some C++ code and diagrams. And let’s dive into C++ linker.

Slide 2.

So, what linker is doing? To answer this question let’s take a look at compilation steps. We have four main stages. And today we will focus on last step. But still lets quickly run through all steps just to remind ourselves about them.

Slide 3.

First we have preprocessing stage. On this stage all preprocessor commands are executed. Like #include, #define and others. So, here is one example. You can see what is the output of this first stage.

But if I will include for example #include <string> it will brings everything inside this header file to our cpp file.

And the first tip comes right away – don’t include everything in your \*.cpp files. It can drastically increase size of your binary files.

But okay, I’m pretty sure that you are aware of it, so let’s move on.

Slide 4.

Ok, second step is translation. So on this stage compiler is working. Lexical analyzer is checking syntax and your file is parsed and divided into tokens. As the output we have assembler listing of your code.

In it you can find the names of function, static variable and some numbers.

Slide 5.

On assembling stage your assembler program is transformed into machine code (binary). Result is stored in binary object files \*.obj. Simply, you can open assembler listing using text editor, but you can’t open object file, because there are only bytes.

But we have dedicated tools for it. Don’t worry we will see a lot of examples of their usage today.

One of such tools is nm. Here is the example of it’s output. As you can see we have name of our function, this constant here…

Slide 6.

So now, let me introduce one more tool that we will use for analysis. It’s name is objdump. And here is one example of it’s output just to give you an intuition of what it can do for you.

-s option shows you the contents of the sections inside object file.

Full description can be found here…

Later we will use this tool with some other options as this is quite a powerful tool.

Slide 7.

Ok, what’s next? I want to remind you that only cpp files are compiled in C++. In other words header files are not compiled standalone and always included in some file with implementation, known as cpp files. And as an output we have object file for each cpp file. But please note that on compilation stage cpp files don’t know anything about each other.

So, consider such example (write on the board):

void foo();

int main()

{

foo();

}

This code is absolutely valid C++, besides the fact it’s not linking. But compilation stage is successful. So, to have it working foo() should be somewhere defined. And if we will combine those 2 \*.cpp files together everything will be just perfect.

Slide 8.

Like in this example.

So, this is exactly what linker is doing. It combines many object files together, building a bridge between them. This link here is this bridge. When those two object files are combined into one executable it’s pretty clear where to go if we want to call function A.

ToDo: add information about reason why linker was introduced (lack of memory). And also that linking is very memory consuming.

Slide 9.

On this slide you can see the linker’s job from other point of view.

In main.o we have A() function undefined (which upper case “U” stands for). But in A.o it is defined according to the upper case letter “T” and it is also defined in the final executable file.

So, we can say that linker is responsible for resolving all undefined symbols from all object files included into the linkage.

So now, I think we can understand why those error messages like “unresolved symbols … smth smth” come from linker.

Slide 10.

As an extension I would like to show you this example.

Usually upper case letters are used for symbols with external linkage and lower case for symbols with internal linkage. B or b means that symbol is uninitialized. d means that symbol is defined. Symbols with no linkage (like localInt) are not considered by linker.

Full description can be found here…

Slide 11.

So far seems easy isn’t it? ☺

So, let’s consider the following example. We have almost the same two cpp files with struct S defined and functions fooA and fooB. Note, that definition of foo methods of structure A in those files are different.

In main we are simply invoking those two functions fooA and fooB.

How do you think what will be the answer?

Slide 12.

Yes, and I can say that this situation is really might happen. As you can see in this example we have output A.cpp twice.

Slide 13.

Ok, now we are just observing the problem… Let’s change the order of linkage of A.o and B.o. Interesting what will be the output?

Slide 14.

And here it is. Now we’ve got B.cpp twice. So, regarding this we can make the hypothesis that when linker found one definition of A it skips all other.

For example I found this problem when I have been writing two different fake classes in UnitTests. I created two derived classes with the same name, from one base class. And overridden their behavior in a different way and put them in global namespace.

And really my first reaction to this was: “why linker is silent here? It won’t be silent if I will define two global functions with the same signatures.”

Of course we will know the answer to this a little bit later, but let’s proceed with the analysis first.

Slide 15.

Here you can see the objdump output with –d options. This option shows you the disassembled output for our binary. As you can see we for both fooA and fooB we call the same implementation of method foo().

Slide 16.

And in this foo() method we use the string A.cpp

If we revert the order of object files linkage than we will see “B.cpp.A.cpp.” here

I want to make on comment to the read only data section. If you will try to remove const qualifier using const\_cast <> of C-style casts, generally it will lead to UB. Because in some platforms this section is protected from write operations. But if you create your variable on the stack, then you can do your cast operations “safely”.

Slide 17.

Ok. So now, let’s see what C++ standard says about it…

One of the reasons of such behavior is the possibility to put the definition of class templates and template functions inside header files. Although there might be some other reasons.

Add explanation of such behavior.

If this example is clear I will move forward…

Slide 18.

I’m almost sure that you heard about static libraries. In general it is just a collection (archive of object files).

On the picture you can see that many object files are combined in one static library. Usually they are named like…

You can use static library as a building block for many of yours applications.

Slide 19.

As shown here. Executable statically link libA.a …

We will look on how to build and use static library just in a few slides.

Not so long time ago one my colleague said that probably linker throw away unused code. I didn’t knew the answer and I decided to clarify it. So, the question to you, how do you think do linker throws away unused code? And what is unused code?

Slide 20.

Ok. Let’s consider example.

We have struct Counter…

Slide 21.

Two cpp files with static instances and a main function that simply prints the counter value.

What you think will be the output?

Slide 22.

2? Do someone has another opinion? Well, very reasonable.

Slide 23.

But now, let’s ship A.o, B.o and Counter.o as a static library. Whoa. And the output now is 0. It is a little bit unexpected.

Add explanation here.

Slide 24.

If we will explicitly put A.o to the linkers input, than result will be one. This will not allow linker to throw unreferenced code away.

Slide 25.

Also there is special option that forces linker to include all object files from static library. But be careful with it, as it can drastically increase the size of your binary. And do not forget to close it with -Wl,-no-whole-archive.

Slide 26.

END OF PART 1.

Slide 27.

So, now we will talk about shared libraries. On windows it is also called dynamic libraries or dll.

Dynamic libraries also can be used like building blocks for different processes but they can be shared in runtime.

First main advantage is memory saving. If we will consider this example we can see that the same version of printf function is included into two executable files. Idea of shared library was to share this function between different processes in runtime and keep only one copy of it.

Second is modularity. It is not so obvious but really important advantage. If we want to change printf we don’t need to recompile all executables that use this function. Once we modify it and rebuild the shared library everyone can use it after restart the application.

And third main advantage is the ability to load and unload library at any moment you want. For example your application can be relatively small and consume little amount of memory and load and unload the dynamic libraries at the time you want to use some functionality.

Slide 28.

Static libraries allow you to call any methods from the libraries you want and C++ syntax allow you to do.

But shared libraries by default do not export all the symbols. In Linux it is done automatically , but not in Windows. In Win you should do it manually.

Win dlls we can consider as some API. You can share those functions that you want.

There are several ways to export symbols. We will consider only one. I’m pretty sure that you will find the rest on the web.

Slide 29.

Shared libraries also can be used similar to static libraries. You can specify whether you want to link it on compile-time or in runtime. We will consider those examples, so don’t worry.

Static linkage means that you want to use shared library during the whole lifetime of your application. It will be loaded with your application and unloaded when your application will finish.

As you can see here we building shared library first. And say to linker that to create static dependency from the shared library.

Please note, that symbols found in shared library are not included into executable. Instead of it name of the symbols are placed. Why? Simply because at compile time we don’t know the exact address of the function inside shared library. It is not even loaded into the memory.

This additional stage is done when loading shared library into the memory. Now we can put the physical addresses from shared memory. In simple words there is additional stage for the linker.

Slide 30.

Let’s consider example of static linking of shared library. Please sorry for a little bit confusing, but from now I will use the terms shared library and dynamic library and will mean the same thing both for Linux and Windows.

We have shared library that has function main\_DLL(). And we have executable with function main().

Slide 31.

To build shared library you don’t need to use ar tool, but you need to specify –shared parameter and link it to your executable in the way similar to static library.

This LD\_LIBRARY\_PATH environment variable specify the path where OS will search for your shared libraries.

Slide 32.

I would like to introduce new tool that you can use for your analysis. It’s name is ldd. It shows all static dependencies of your application. If you such error…

Slide 33.

Ok, so now let’s look on dynamic linkage.

Dynamic linkage means that you want to use shared library only during some period of the lifetime of your application.

Here you can see that there is no dependency between those components on the compilation stage. And all dependencies are visible only in the code.

We can manipulate loading, using and unloading of the library from the code.

Slide 34.

Here we should add extern C qualifier for our function inside shared library. Because in C++ the symbol names are mangled and we won’t do that we won’t be able to call this function. I will just say few words, that this qualifier force compiler not to decorate names of our symbols.

More information you can find on the web.

Slide 35.

And actually here is the code for loading library and invoking main\_DLL() from it.

Slide 36.

And here is the compilation steps that we need to make. As you can see we only loading ld which includes functions like dlopen() and dlsym().

Slide 37.

To give you an intuition how it is working and what you need to keep in mind let’s consider following example. This example is from my own experience and yes I did spent quite some time on fixing this problem.

So,We have struct that count it’s instances. This code should be very clear to you.

Slide 38.

We have shared library that has the function that create one instance of this class and prints the address of the static counter and its value.

Slide 39.

Ok. And we have an executable that call main\_DLL function and repeat those actions once again.

Slide 40.

Let’s build and launch it on Linux. And we can see the same counter is addressed from both DLL and executable.

Slide 41.

Let’s prove that output by doing a little analysis once again. We can see that both symbols have V class. According to … And we can see the address of the static member and we can conclude that this member is inside executable.

If it is clear than we will move forward.

Slide 42.

Let’s now see what will be on Windows. We need to use those \_\_declspec qualifiers with export on DLL side and import on executable side.

Slide 43.

If we will build it and launch on Win we will see this a little bit surprising result.

We see that those static members are different for DLL and executable.

Slide 44.

Let’s do analysis once again. And we see that both in DLL and executable those static members have class D which stands for defined. And also we can spot the addresses of them.

Slide 45.

So, this is the schematic view of what is going on in our situation. We have two static counters, one came from DLL and another one is let’s say our own counter. I showed them both here, because the counter that comes from DLL is not shared between different processes. Only main\_DLL is shared.

And on the picture below you can see desired situation when we have only one counter, like in Linux case.

Slide 46.

Let me just draw the same picture in a little bit different manner. I want to show you the source of those static members. Note that those components are built separately, that’s why they creating their own copies of static members.

Slide 47.

What you think how we can solve this problem?

What we want to do is to break the link between static counter and executable and export this function from DLL. With this approach executable will not place one instance of static member but will imports the one from shared library.

Slide 48.

So, first we split declaration and definition of Counter. And create special macros that we will use further. On dll side it will expand like export, on executable like import. This is an absolutely normal way to process export/import symbols.

Slide 49.

And we export counter class and main\_DLL from DLL using this macros.

Slide 50.

On executable side we are importing counter class and main\_DLL using the same macros.

Slide 51.

And we received the result that we want.

Slide 52.

To make two different counters on Linux I haven’t found any other way but to make such a “hack”. You need to redefine the name of the class. Thus you will force compiler to create different counters, because now there are two different classes.

But generally this problem could have more elegant solution. We can transform shared library into static and link them like that (draw the picture on the board).

Slide 53.

In conclusion I would like to share my preferable structure of small projects…

As far as shared library is loaded into the shared memory everything that is inside data segment of the shared library goes into the process address space. That’s why I show it here, not here. Shared memory cannot share the data because it will mean that process can modify shared data, but it’s not true. OS would not appreciate such way of interprocess communication, right? Processes can only modify it’s own address space.

Maybe find another name for hereditary disease and static member mess up

Добавить к слайду про анализ вывода nm для static member mess up :: unix version объяснение как раскрывается данная переменная во время загрузки dll.