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.

Tell about ldd. In article on habr.

Maybe find another name for hereditary disease

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