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.

Say few words about .rodata section. UB when you try to const\_cast it.

Add the link to C++ standard about one definition rule.

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.