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!

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”. Even though it will be good to know about possible traps that we can get in, right?

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 command 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.

You maybe already know what I want to say? Yes, this is first tip – don’t include everything in your \*.cpp files. It can drastically increase size of your binary files.

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.

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.

Ok, what’s next? I want to remind you that only cpp files are compiled in C++. And as an output we have object file for each cpp file. So, 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

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

So, this is exactly what linker is doing. It combines many object files together.

Slide 7.

Because if we will look at such example:

static int a = 1234;

static double doub;

char chr;

void A()

{

int b = a;

(void)b;

(void)chr;

double d = doub;

(void)d;

}

nm -A -C -S A.o

A.o:0000000000000000 000000000000001a T A()

A.o: U \_\_gxx\_personality\_v0

A.o:0000000000000000 0000000000000004 d a

A.o:0000000000000000 0000000000000001 B chr

A.o:0000000000000008 0000000000000008 b doub

void A();

int main()

{

A();

}

We see that A() and B() have U class. This means that those symbols are not defined yet. It is blank space, that is supposed to be resolved in future. We hope that we will find it in some object file. But as you can see here we have also T. It means that this is defined function with external linkage, in other words global function. Functions that have internal linkage has class t.

Slide 8.

Tell about ldd. In article on habr.