Лекция 1

к будущему себе, пришлось делать в либре, все остальное не очень подходило.

Когда мы говорим о тулчейне, мы говорим о

- Системах компиляции
- Компиляторе
- Ассемблере
- Линкере
- Отладке (gdb)
- Производительности
- Портирование компилятора (?)

gcc -O2 main.cpp -o main.x *Что тут происходит* ?
Что делает gcc, чтобы из кода cpp получился исполняемый файл .x ?

1. Запуск компилятора, компилятор перевод с языка исходного кода
на язык ассемблера

2. Запуск ассемблера, который ассмблирует программу (?)

3. Запуск линкера, который линкует

4. Программа становится исполняемой

Совсем простой **main**

```
//#include <iostream>

int main() {

  return 9;
}
```

gcc -O2 main.cpp -o main.x
gcc -O2 main.cpp -o main.x --verbose

1. *Запуск компилятора*

Можно увидеть

/usr/lib/gcc/x86_64-linux-gnu/11/**cc1plus -quiet** -v
-imultiarch x86_64-linux-gnu -D_GNU_SOURCE main.cpp
-quiet -dumpdir main.x- -dumpbase main.cpp -dumpbase-ext
.cpp **-mtune=generic -march=x86-64 -O2** -version -fasynchronous-unwind-tables
-fstack-protector-strong -Wformat -Wformat-security -fstack-clash-protection -fcf-protection
-o /tmp/ccVTKgpx.s

До этого печатается служебная информация о том, как компилятор был собран

cc1plus <-- компилятор для C++
cc1 <-- для C было бы так

-mtune=generic <-- не применять специфичных оптимизаций, скомпилируй для любого x86,
например можно подать опцию оптимизации для специфичных процессоров типа Skylake

-march=x86-64 - артитектура под которую мы генерируем код

Результат работы компилятора это ассемблерный файл например вот такой /tmp/ccNc7xfT.,
имя выбрано случайно

2. *Запуск ассемблера*

as -v --64 -o /tmp/ccGK9lsp.o /tmp/ccNc7xfT.s
GNU assembler version 2.38 (x86_64-linux-gnu) using BFD version (GNU Binutils for Ubuntu) 2.38

Причем вот это /tmp/ccNc7xfT.s и есть результат работы компилятора, а не main.x :)
В tmp компилятор кладет временные файлы, потом все промежуточные файлы удаляет.

В данной команде as -v --64 -o /tmp/ccGK9lsp.o /tmp/ccNc7xfT.s
видно, что ассемблер сделал из ассемблерное файла /tmp/ccNc7xfT.s
объектный файл /tmp/ccGK9lsp.o

3. *Запуск линкера*

COLLECT_GCC_OPTIONS='-O2' '-o' 'main.x' '-v'
'-shared-libgcc' '-mtune=generic' '-march=x86-64' '-dumpdir' 'main.x.'

/usr/lib/gcc/x86_64-linux-gnu/11/collect2

/usr/lib/gcc/x86_64-linux-gnu/11/**collect2** -o main.x <-- запуск линкера

ld запускается из под **collect2**

ld это gnu линкер

g++ -O2 main.cpp -o main.x --verbose -save-temps <-- работа компилятора с
сохранением промежуточных файлов

```
$ ls -lh
total 32K
-rw-rw-r-- 1 fima fima   51 янв       5 03:22 main.cpp
-rwxrwxr-x 1 fima fima  16K янв       5 03:44 main.x
-rw-rw-r-- 1 fima fima  160 янв       5 03:44 main.x-main.ii  ← создал Препроцессор (g++ -E)
-rw-rw-r-- 1 fima fima 1,3K янв       5 03:44 main.x-main.o  ← создал Ассемблер
-rw-rw-r-- 1 fima fima  479 янв       5 03:48 main.x-main.s  ←создал Компилятор
```

Обращаем внимание объектный файл main.x-main.o занимает 1.3K,
а исполняемый main.x занимает 16K, откуда взялись 15K ???

Видимо, мы подключили какую run-time библиотеку.

И тут вопрос, а как мы пришли в main ???

15K нужны именно для того, чтобы прийти в main;

// ==

Входной код, переводится во внутрненее представление компилятора (IR),
над IR делаются уже оптимизации.

Оптимизации это парадно-выходное слово :)

То что делает компилятор корректно называется трансформацией. Трансформация необязательно делает код лучше, она в среднем делает его лучше, а может и хуже, как повезёт.

Процесс подачи кода в компилятор это frontend компилятра.

Компилятор на своем внутреннем представлении делает трансформации, делает.s файл в процессе кодагенерации. Это еще называется backend компилятора.

HIR - Hight Level Intermidiate Representation

-> Frontend <-

Препроцессинг
Лексический анализ
Синтаксический анализ
Семантический анализ
Построение HIR

-> Middleend + Backend <-

Оптимизации HIR
Оптимизации MIR
Оптимизации LIR
Кодогенерация

Что такое человек ?

Это часть аппаратуры, которая печатает, когда вы ждёте стандартный ввод.

----------**Препроцессинг**----------

Немного меняем наш main

```
#include <iostream>

int main() { std::cout << "hello ! \n"; return 0;}
```

g++ -E main.cpp -o main.i

Сколько строк будет в main.i ?

На первый взгляд у нас тут вообще однострочник, да, но там есть
еще здоровенный include. Но насколько здоровенный ?

main.i это результат выполенения фазы препроцессинга, то есть фазы лексического анализа. Компилятор будет обрабатывать все, что осталось после препроцессирования.

$ **wc -l main.i**
32260 main.i

Препроцессирование чудовищно раздувает код.

ls -lh
-rw-rw-r-- 1 fima fima   76 янв 5 05:08 main.cpp
-rw-rw-r-- 1 fima fima 761K янв 5 05:09 main.i
-rwxrwxr-x 1 fima fima 16K янв 5 03:44 main.x

Важно сказать, что оптимизации никак не влияют на результат, потому что оптимизации это middle end, а препроцессинг это фаза frontend.

----------**Синтаксический анализ**----------

У нас есть поток лексем, который мы получаем из кода C, C++ и мы должны сделать синтаксическое дерево из этих лексем.

Синтаксический анализ имеет свои детали. Например построение синтаксического дерева не всегда однозначно.

Пример. Что здесь написано ?

```
std::ifstream datafile ("ins.dat");
std::list<int> data (std::istream_iterator<int>(datafile), std::istream_iterator<int>());
```

Вроде бы как конструируем список из двух итераторов, но есть один момент синтаксического анализа.
--> Здесь **Most vexing parse**.
С точки зрения анализа тут написано **объявление функции**.

Функция data, которая возвращает std::list<int> и принимает два параметра

1. Функцию, которая принимает datafile и возвращает is std::istream_iterator
2. Функцию, которая ничего не принимает и возвращет std::istream_iterator

!!!
Это особенности грамматического разбора.

В грамматическом разборе C++,

1. Все что, может быть засчитано как поле класса, будет засчитано как поле класса, поэтому мы пишем **typename** в шаблоне.

2. Все, что может быть засчитано как арифметический оператор, засчитывается, как арифметический оператор поэтому мы пишем **template <typename T>**

3. Все, что может быть защитано быть как объявление функции, будет засчитано, как объявление функции. Поэтому мы используем фигурные скобки или двойные круглые.

```
std::list<int> data {std::istream_iterator<int>(datafile), std::istream_iterator<int>()};
```

Так же хочу привести кусок кода, по поводу итераторов.

```
std::ofstream output_file("testing spaces.txt");
output_file << " 1" << " 222 222     " << " 7 a 7   7" << "999" << std::endl;
output_file << " 4" << " 5687     " << std::endl;


std::ifstream input_file("testing spaces.txt");
std::vector<int> test{std::istream_iterator<int>(input_file), std::istream_iterator<int>()};
```

Важно понимать, что если во время чтения входного потока и будет встречено что то, отличающееся от int. То последующее чтение ввода будет прекращено. **То есть "1", "222", "222" и "7" попадут в лист, весь остальной ввод будет отброшен.**

Пример 2. Тоже самое

```
template <typename T> foo (T x) {
        T::iterator *y; <--- Будет воспринято как умножение, фиксится через typename
        /* .... */
}
```

Макрос это правило для переопределения лексем.

Таким образом программист может вмешиваться в Синтаксический анализ.