Язык C++

Компиляция

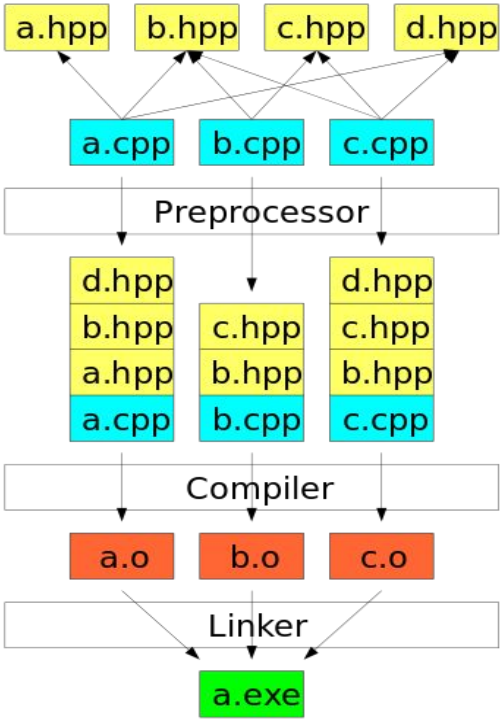
Программы состоят из файлов

- Логическое разбиение на модули
- Повторное использование
- Ускорение процесса компиляции
- Небольшие файлы проще читать и редактировать
- Есть два типа файлов
  - Заголовочные (*.h , *.hpp)
  - С исходным кодом (*.cpp)

Этапы трансляции

1. Препроцессор
2. Компиляции
3. Линковщик

NB! На самом деле этапов 9



Этапы трансляции

Компиляторы умеют запускать этапы трансляции по отдельности (на примере clang):

- **clang++ -E** - run preprocess,
- **clang++ -S** - run preprocess and compilation steps
- **clang++ -c** - run preprocess, compile, and assemble steps
- **clang++ --Xlinker** - run linker

Объявление и определение

Declaration

- Задает имя и прочие атрибуты для сущностей (например сигнатуру функции)
- Сколько угодно раз

Definition

- Полностью определяет сущность
- Является одновременное объявлением

Объявление и определение

```
// declaration
int add(int a, int b);

//definition
int add(int a, int b) {
   return a + b;
}
```

Объявление и определение

```cpp
// math.cpp

//definition
int add(int a, int b) {
   return a + b;
}
```

```cpp
// main.cpp

int add(int a, int b);

int main() {
   int i = add(10, 2);
   return 0;
}
```

Заголовочный файл

- Function declaration & definition
- Variable declaration
- Type declaration
- Static variable definition & declaration
- …

Заголовочные файлы

```
// math.h

#pragma once

int add(int a, int b);
```

Заголовочные файлы

```cpp
// math.h

#pragma once

int add(int a, int b);
```

```cpp
// main.cpp

#include "math.h"

int main(int, char**){
    int c = add(10, 2);
    return 0;
}
```

Препроцессор

- Язык препроцессора – это специальный язык программирования, встроенный в C++.
- Лексический анализ кода.
- Команды языка препроцессор называют директивами, все директивы начинаются со знака #.
- Директива #include позволяет подключать заголовочные файлы к файлам кода.
- Препроцессор заменяет директиву #include "bar.h" на содержимое файла bar.h.

Препроцессор

1. **#define и #unded (__FILE__, __LINE__, …)**
2. **# and ## operators**
3. Условное включение(**#if #ifdef #ifndef #elif #else #endif)**
4. **#include**
   a. **#include <***filename***>**
   b. **#include "***filename***"**
5. **#pragma once**
6. **#error**
7. **etc**

#include

```
#include "math.h"

int main(int, char**){
    int c = add(10, 2);
    return 0;
}
```

#include

clang++ -E main.cpp

```
# 1 "main.cpp"
# 1 "<built-in>" 1
# 1 "<built-in>" 3
# 537 "<built-in>" 3
# 1 "<command line>" 1
# 1 "<built-in>" 2
# 1 "main.cpp" 2
# 1 "./math.h" 1


int add(int a, int b);
# 2 "main.cpp" 2

int main(int, char**){
    int c = add(10, 2);
    return 0;
}
```

Макросы

- Макросами в C++ называют инструкции препроцессора.
- Препроцессор C++ является самостоятельным языком, работающим с произвольными строками.
- Макросы можно использовать для определения функций:

Препроцессор “не знает” про синтаксис C++.

Макросы

```cpp
#include <iostream>

#define MAX(x, y) (x > y ? x : y)

int main() {
  // Some comment
  std::cout << MAX(10, 20) << std::endl;

  return 0;
}
```

Макросы

clang++ -E main.cpp

```
content of iostream

# 2 "main.cpp" 2

int main() {
    std::cout << (10 > 20 ? 10 : 20) << std::endl;

    return 0;
}
```

Макросы

```
#ifdef __DEBUG__
   #define error_log(format, ...) printf("[ERROR] (%s:%d) " format "\n", __FILE__,
__LINE__, ##__VA_ARGS__)
#else
   #define error_log(format, ...)
#endif

int main() {

   error_log("fatal errro no: %d", 1);

   return 0;
}
```

Макросы

```
#define log(type, format, ...)  printf("[" type "] (%s:%d) " format "\n", __FILE__,
__LINE__, ##__VA_ARGS__)
```

```
#define log_error(...) log("ERORR", ##__VA_ARGS__ )
#define log_info(...) log("INFO", ##__VA_ARGS__ )
```

Стражи

```
#ifndef HEADER_MATH_H
#define HEADER_MATH_H

int add(int a, int b);

#endif
```

```
#pragma once

int add(int a, int b);
```

pragma pack

```
#pragma pack(1)
struct SomeStruct {
   int i;
   char ch;
};
#pragma pack


int main() {

   printf("%d\n", sizeof(struct SomeStruct));
   return 0;
}
```

Компилятор

- На вход компилятору поступает код на C++ после обработки препроцессором.
- Каждый файл с кодом компилируется отдельно и независимо от других файлов с кодом.
- Компилируется только файлы с кодом (т.е. *.cpp).
- Заголовочные файлы сами по себе ни во что не компилируются, только в составе файлов с кодом.
- На выходе компилятора из каждого файла с кодом получается “объектный файл” — бинарный файл со скомпилированным кодом (с расширением .o или .obj).

Компилятор

```cpp
// math.cpp

void increment(int& v) {
    v++;
}

int add(int a, int b) {
    return a + b;
}
```

```cpp
// main.cpp

int main(int, char**) {
    int x = 1;
    int y = 2;

    increment(y);
    int result = add(x, y);

    return 0;
}
```

```
        .text
        .globl  _Z9incrementRi          # -- Begin function _Z9incrementRi
        .p2align4, 0x90
        .type   _Z9incrementRi,@function
_Z9incrementRi:                         # @_Z9incrementRi
        .cfi_startproc
# %bb.0:
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset %rbp, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register %rbp
        movq    %rdi, -8(%rbp)
        movq    -8(%rbp), %rax
        movl    (%rax), %ecx
        addl    $1, %ecx
        movl    %ecx, (%rax)
        popq    %rbp
        .cfi_def_cfa %rsp, 8
        retq
.Lfunc_end1:
        .size   _Z9incrementRi, .Lfunc_end1-_Z9incrementRi
        .cfi_endproc
                                # -- End function
```

```
main:                                   # @main
        .cfi_startproc
# %bb.0:
        pushq%rbp
        .cfi_def_cfa_offset 16
        .cfi_offset %rbp, -16
        movq %rsp, %rbp
        .cfi_def_cfa_register %rbp
        subq  $48, %rsp
        movl  $0, -4(%rbp)
        movl  %edi, -8(%rbp)
        movq %rsi, -16(%rbp)
        movl  $1, -20(%rbp)
        movl  $2, -24(%rbp)
        leaq   -24(%rbp), %rdi
        callq  _Z9incrementRi
```

Компоновщик(линкер)

- На этом этапе все объектные файлы объединяются в один исполняемый (или библиотечный) файл.
- При этом происходит подстановка адресов функций в места их вызова.
- По каждому объектному файлу строится таблица всех функций, которые в нём определены.

Компоновщик(линкер)

```cpp
// math.h
#pragma once

void increment(int& v);
int add(int a, int b);
```

```cpp
// math.cpp
#include "math.h"
void increment(int& v) {
    ++v;
}

int add(int a, int b) {
    return a + b;
}
```

```cpp
#include "math.h"
#include <iostream>

int main(int, char**) {
    int x = 1;
    int y = 2;

    increment(y);
    int result = add(x, y);

    std::cout << result << std::endl;

    return 0;
}
```

Linkage

- **External** - доступность из всех единиц трансляции
- **Internal** - доступность из текущей единицы трансляции
- **No linkage** - только текущий скоуп

Storage duration

- **automatic** - время жизни ограничено скоупом объявления
- **static** - время жизни от запуска программу и до ее окончания
- **thread** - время жизни ограничено потоком
- **dynamic** - new\delete

Storage class specifier

- **static** - static duration and internal linkage
- **extern** - static duration and external linkage
- **thread_local** - thread storage duration
- **mutable**

Storage duration and linkage

```
// math.h
#pragma once

const static float PI = 3.14f;
const extern float Exp;
const int SomeValue = 239;
static float PI2 = 3.14f;
extern float Exp2;
// int SomeValue2 = 239; // error: multiple definition of `SomeValue2';

void PrintInfo();
```

Storage duration and linkage

```cpp
// math.cpp
#include "math.h"
#include <iostream>

void PrintInfo() {
   std::cout << PI << " Address of PI:" << &PI << std::endl;
   std::cout << Exp << " Address of Exp:" << &Exp << std::endl;
   std::cout << SomeValue << " Address of Exp:" << &SomeValue << std::endl;
   std::cout << PI2 << " Address of PI:" << &PI2 << std::endl;
   std::cout << Exp2 << " Address of Exp:" << &Exp2 << std::endl;
}
```

Storage duration and linkage

```
#include "math.h"
#include <iostream>
const float Exp = 2.72f;
float Exp2 = 2.72f;

int main(int, char**) {
   PI2 = 3;
   Exp2 = 2;
   std::cout << PI << " Address of PI:" << &PI << std::endl;
   std::cout << Exp << " Address of Exp:" << &Exp << std::endl;
   std::cout << SomeValue << " Address of Exp:" << &SomeValue << std::endl;
   std::cout << PI2 << " Address of PI:" << &PI2 << std::endl;
   std::cout << Exp2 << " Address of Exp:" << &Exp2 << std::endl;
   std::cout << "----Math-----\n";
   PrintInfo();
   return 0;
}
```

Storage duration and linkage

```
3.14 Address of PI:0x402018
2.72 Address of Exp:0x402014
239  Address of Exp:0x40201c
3    Address of PI:0x404074
2    Address of Exp:0x404070
----Math-----
3.14 Address of PI:0x402050
2.72 Address of Exp:0x402014
239  Address of Exp:0x402054
3.14 Address of PI:0x404078
2    Address of Exp:0x404070
```

Ошибки компиляции

- Ошибки компиляции
- Ошибки линковки