1 Система трансляции программ в Unix

1.1 Схема трансляции программы

Рассмотрим схему трансляции программы на языке Си, которая традиционно используется в системах Unix. Трансляция программы состоит из следующих этапов:

1. препроцессирование;
2. трансляция в ассемблер;
3. ассемблирование;
4. компоновка.

Традиционно исходные файлы программы на языке Си имеют суффикс имени файла `.c`, заголовочные файлы для программы на Си имеют суффикс `.h`. В файловых системах типа Unix регистр букв значим, и если, например, имя файла имеет суффикс `.C`, такой файл считается содержащим текст программы на языке Си++, и будет компилироваться компилятором языка Си++, а не Си.

Препроцессирование. Препроцессирование уже было рассмотрено нами ранее. Препроцессор просматривает входной `.c` файл, исполняет в нём директивы препроцессора, включает в него содержимое других файлов, указанных в директивах `#include` и пр.

В результате получается файл, который не содержит директив препроцессора, все используемые макросы раскрыты, вместо директив `#include` подставлено содержимое соответствующих файлов. Файл с результатом препроцессирования обычно имеет суффикс `.i`, однако после завершения трансляции все промежуточные временные файлы удаляются, поэтому такой файл, как правило, никогда не виден пользователю. Результат препроцессирования называется *единицей трансляции*.

Трансляция в ассемблер. Это — основная фаза работы. На вход подаётся одна единица трансляции, а на выходе (при отсутствии синтаксических и семантических ошибок) выдаётся файл на языке ассемблера для (как правило) машины, на которой ведётся трансляция. Файл с оттранслированной программой на языке ассемблера имеет суффикс имени `.s`, но точно так же, как и результат работы препроцессора, он, как правило, не виден пользователю.

Ассемблирование. На этой стадии работает ассемблер. Он получает на входе результат работы предыдущей стадии и генерирует на выходе объектный файл. Объектные файлы традиционно имеют суффикс `.o`. Программа-ассемблер в системах Unix обычно называется as.

Компоновка. Компоновщик получает на входе объектные файлы для каждой единицы трансляции, из которых состоит программа, подключает к ним стандартную библиотеку языка Си и библиотеки, указанные пользователем, и на выходе получает исполняемую программу. В системах Unix исполняемые двоичные программы не имеют никакого специального суффикса, например, оболочка-драйвер для компилятора GNU C называется просто gcc. Компоновщик (редактор связей) в системах Unix обычно называется ld.

1.2 Запуск транслятора gcc

Рассмотрим основные возможности транслятора GNU C. Транслятор запускается командой `gcc <files-and-options>`. В командной строке задаётся список файлов, которые должны быть оттранслированы и объединены в один исполняемый файл. Какие операции необходимо выполнить с файлом — зависит от суффикса имени файла. Возможные

суффиксы перечислены в таблице 1. Если имя файла имеет нераспознанный суффикс, это имя передаётся компоновщику.

`.h`	Заголовочный файл на языке Си. Попытка трансляции такого файла вызывает сообщение об ошибке.
`.c`	Файл на языке Си. Выполняется препроцессирование, трансляция, ассемблирование и компоновка.
`.i`	Препроцессированный файл на языке Си. Выполняется трансляция, ассемблирование и компоновка.
`.s`	Файл на языке ассемблера. Выполняется ассемблирование и компоновка.
`.S`	Файл на языке ассемблера. Выполняется препроцессирование, ассемблирование и компоновка.
`.o`	Объектный файл. Выполняется компоновка.
`.a`	Файл статической библиотеки. Выполняется компоновка.

Таблица 1: Суффиксы имён файлов для транслятора **gcc**

Набор действий определяется для каждого файла индивидуально. Например, если в командной строке указаны имена файлов `1.c` и `2.o`, то для первого файла будут выполнены все шаги трансляции, а для второго — только компоновка. Исполняемый файл будет содержать результат трансляции первого файла, скомпонованный со вторым файлом и стандартными библиотеками.

Пользователь может явно задать, на такой фазе нужно остановиться. По умолчанию транслятор пытается выполнить все необходимые фазы, включая компоновку программы. Конечная фаза трансляции программы определяется для всех транслируемых за один вызов **gcc** файлов указанием одной из опций, перечисленных в таблице 2.

`-E`	Остановиться после препроцессирования. Результат работы препроцессора выводится по умолчанию на стандартный поток вывода. Имя выходного файла можно указать с помощью опции `-o`. При этом если в командной строке указано несколько файлов, то в выходной файл будет помещён результат препроцессирования последнего файла.
`-S`	Остановиться после трансляции в ассемблер. По умолчанию имя выходного файла получается из имени входного файла заменой суффикса `.c` или `.i` на суффикс `.o`. Явное имя выходного файла можно указать с помощью опции `-o`. Попытка использования опции `-o` и нескольких имён входных файлов вызывает сообщение об ошибке.
`-c`	Остановиться после ассемблирования. По умолчанию имя выходного файла получается из имени входного файла заменой суффикса его имени на суффикс `.o`. Явное имя выходного файла можно указать с помощью опции `-o`, которая несовместима с указанием одновременно нескольких транслируемых файлов. Если ни одной из перечисленных выше опций не задано, выполняются все стадии трансляции. Имя выходного файла по умолчанию равно `a.out`, но может быть изменено с помощью опции `-o`.
`-o`	Позволяет задать явное имя выходного файла для любой стадии трансляции.

Таблица 2: Опции конечной фазы транслятора **gcc**

Например, командная строка

```
gcc 1.c 2.c -o 1
```

транслирует два файла на языке Си, объединяя их в одну программу с именем `1`. Командная строка

```
gcc 3.o 4.o -o 3 -lm
```

компонует два объектных файла, добавляя к ним стандартную библиотеку языка Си и стандартную математическую библиотеку (опция `-lm`), и помещает результат в исполняемый файл с именем `3`.

Прочие полезные опции транслятора **gcc** перечислены в таблице 3.

1.3 Компоновка программы

Если исполняемая программа компонуется из нескольких единиц трансляции, компоновщик использует свои правила видимости имён, которые приведены ниже.

- Все имена, объявленные с классом памяти **`static`**, видимы только в пределах своей единицы трансляции и не влияют на компоновку.
- Если некоторая единица трансляции использует внешнее имя (переменной или функции), которое не определено ни в какой единице трансляции, выдаётся сообщение об ошибке.
- Если несколько единиц трансляции определяют нестатическую функцию с одним и тем же именем, выдаётся сообщение об ошибке.
- Если некоторое нестатическое имя определяется и как переменная, и как функция, выдаётся сообщение об ошибке.
- Если несколько единиц трансляции определяют нестатическую инициализированную переменную с одним и тем же именем, выдаётся сообщение об ошибке.
- Если несколько единиц трансляции определяют переменную с одним и тем же именем, которая инициализируется не более чем в одной единице трансляции, все определения размещаются, начиная с одного адреса (класс памяти `common`).

Последнее правило можно продемонстрировать на следующем примере. Предположим, что в трёх файлах определена переменная `var` следующим образом:

```
int var = 1;
int add1(void)
{
  return var++;
}
```

```
int var;
int add2(void)
{
  return var += 2;
}
```

```
int var;
int add3(void)
{
  return var += 3;
}
```

Если все три единицы компиляции объединяются в одну программу, то переменная `var` каждого из трёх файлов будет располагаться по одному и тому же адресу, и каждая их трёх функций будет работать, по сути, с общей переменной. Чтобы предотвратить такое слияние переменных можно использовать явную инициализацию переменной `var` нулём, тогда компоновщик выдаст сообщение об ошибке.

`-I PATH`	Добавляет каталог `PATH` в начало списка каталогов, которые просматриваются препроцессором при поиске файлов, подключаемых директивой `#include`. В командной строке может быть указано несколько опций `-I`, тогда каталоги просматриваются в порядке, в котором они указаны в командной строке.
`-D NAME`	Определяет макрос с именем `NAME`, который получает значение 1.
`-D NAME=VALUE`	Определяет макрос с именем `NAME`, который получает заданное значение.
`-Wall`	Включает выдачу большого количества предупреждающих сообщений, которые по умолчанию не выдаются. Опция должна использоваться при компиляции программ, все предупреждающие сообщения компилятора должны быть внимательно проанализированы, поскольку сообщения могут указывать на ошибки в программе.
`-g`	Включает генерацию отладочной информации в исполняемую программу. Наличие отладочной информации позволяет отлаживать программу в терминах исходного языка, а не машинного кода.
`-O2`	Включает большинство оптимизаций программы, которые одновременно уменьшают размер программы и увеличивают скорость её выполнения.
`-L PATH`	Добавляет путь `PATH` в начало списка каталогов, которые просматриваются редактором связей при поиске библиотек, указанных с помощью опции `-l`. Если в командной строке указано несколько опций `-L`, они добавляются в том же порядке, в котором указаны в командной строке.
`-lname`	Добавляет библиотеку `name` к списку библиотек, которые участвуют в компоновке программы (обратите внимание на отсутствие пробела между опцией и именем библиотеки). В системах Unix редактор связей просматривает библиотеки *один раз*, поэтому неправильный порядок задания библиотек может привести к тому, что некоторые имена останутся неопределёнными, и компиляция завершится с ошибкой. Файл, хранящий библиотеку с именем `name`, называется `libname.a`, если библиотека статическая, и `libname.so`, если библиотека динамическая.
`-static`	Указывает, что при компоновке не должны использоваться динамические библиотеки. Реализации всех используемых в программе функций будут добавлены непосредственно в исполняемый файл. Это может привести к тому, что размер тривиальной программы вырастет до сотни килобайт, зато такая программа перестанет быть зависимой от динамических библиотек, и на некоторых системах только статически скомпонованные программы могут отлаживаться.

Таблица 3: Прочие опции транслятора gcc

1.4 Программы из нескольких единиц трансляции

Только самые простые программы размещаются полностью в одном исходном файле. Более сложные программы состоят из нескольких исходных файлов, которые объединяются

компоновщиком. При написании таких программ полезно следовать следующим рекомендациям.

- При группировке функций и переменных по исходным файлам логически сильно связанные функции объединяются в один исходный файл. Например, функции работы с файлом таблицы могут быть помещены в один исходный файл, функции, которые выводят на экран содержимое таблицы, — в другой файл, в функции, которые анализируют ввод пользователя, — в третий файл.

- Чем больше переменных объявлено в единице компиляции с классом памяти **`static`** вместо класса памяти по умолчанию `common`, тем лучше. Лучше всего, если доступ к данным всегда происходит с помощью вызовов функций. Чем меньше «чужих» переменных использует некоторая единица компиляции, тем она проще для понимания.

- Для каждого `.c` файла должен существовать интерфейсный файл с тем же именем, но суффиксом `.h`, в котором определяются переменные, функции, типы данных и пр., которые могут использоваться извне данной единице компиляции.

- Исходный `.c` файл должен обязательно подключать свой собственный `.h`-файл. В этом случае транслятор обнаружит рассогласования между объявлениями в `.h`-файлах и определениями в `.c`-файле.

- Интерфейсный `.h` файл должен быть обязательно защищён от повторного включения следующей конструкцией:

  ```
  #ifndef __NAME_H__
  #define __NAME_H__
  <здесь находится текст файла>
  #endif
  ```

  Здесь `NAME` — это имя файла (без суффикса). Поскольку некоторые `.h`-файлы могут включать другие `.h`-файлы, когда программа становится большой, человек уже не может отследить, какие файлы уже включались, а какие — ещё нет. Поэтому в `.c` файле включаются все заголовочные файлы, необходимые данной единице компиляции. Защита от повторного включения предотвращает появление ошибок о переопределённых типах, переменных и функциях.

- В заголовочном файле помещаются макроопределения и типы данных, являющиеся интерфейсом данной единицы компиляции, то есть необходимые для использования функций и переменных этой единицы компиляции. С классом памяти **`extern`** помещаются необходимые переменные и прототипы функций, объявленные в соответствующей единице компиляции.

- В заголовочный файл никогда не помещаются тела функций и определения переменных с классом памяти, отличным от класса **`extern`**. В заголовочный файл никогда не помещаются прототипы функций с классом памяти **`static`**. Если некоторый тип или константа используются только в теле какой-либо функций и не нужен для правильной работы с функциями и переменными данной единицы компиляции, этот тип или константа также не помещаются в заголовочный файл.

1.5 Inline-функции

Стандарт C99 позволяет определять так называемые inline-функции, то есть функции, тело которых подставляется в точки вызова. Такие функции по эффективности не уступают макросам, но не имеют их отрицательных сторон, рассмотренных ранее (то есть неверных приоритетов операций, многократного вычисления побочных эффектов и пр.). Для объявления inline-функций используется ключевое слово **inline**.

```
1 inline int max(int a, int b)
2 {
3   return a > b ? a : b;
4 }
```

По умолчанию inline-определение функции не является внешним определением, то есть не будет видна из других единиц компиляции. Другие единицы компиляции могут содержать свои inline-определения функции `max`. Кроме того, некоторая единица компиляции **должна** содержать внешнее определение функции `max`, то есть определение, не содержащее ключевых слов **inline** или **static**.

Чтобы указать, что некоторое inline-определение функции является одновременно и внешним определением, можно в той же единице компиляции разместить прототип функции с классом памяти **extern**.

```
extern int max(int a, int b);
```

Если даже в единице компиляции и содержится inline-определение некоторой функции, компилятор может проигнорировать его и в точку вызова данной функции вставить обычный вызов внешней функции.