7. СРЕДСТВА РАЗРАБОТКИ ПРОГРАММНАЯЗЫКЕ ПРОГРАММИРОВАНИЯ СИ

7.1 Компиляция и связь программных модулей

Система разработки программ на языках программирования Си/Си++ состоит из препроцессора, компилятора, ассемблера и редактора связей. Также к средствам разработки можно отнести библиотеки функций с сопутствующими заголовочными файлами и справочную информацию по ним.

Наиболее популярным средством для транслирования текстов программ в исполняемый файл является GNU Compiler Collection (**gcc**) – это коллекция компиляторов различных языков программирования (C, C++, Objective-C, Java, Fortran, Ada) и сопутствующие средства: ассемблер, препроцессор, редактор связей и т. п.

Схематично процесс трансляции можно разбить на следующие стадии (рис. 7.5):

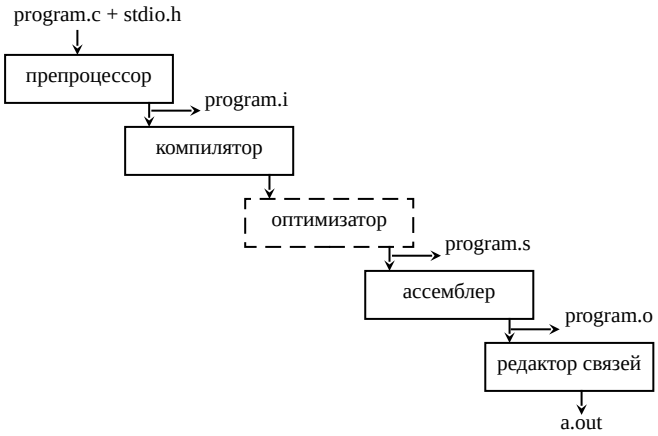

Рис. 7.5. Процесс генерации исполняемого файла по исходному тексту

Первая стадия. Исходный текст программы на языке Си (суффикс «.c») или Си++ (суффикс «.C», «.cpp»), а также заголовочные файлы (суффикс «.h») обрабатываются *препроцессором*. На этой стадии в тексте программы интерпретируются такие директивы, как *#define*, *#include* и т. п. Полученный результат (текст программы, не требующий препроцессорной обработки) обычно имеет суффикс «.i» и без явного указания не создаётся.

Вторая стадия. *Компилятор* Си/Си++ производит синтаксический разбор текста и формирует соответствующий ему ассемблерный код. После чего возможна обработка текста *оптимизатором*, позволяющим сократить размер получаемого кода и увеличить скорость его работы. Полученный результат (текст программы на языке ассемблера) обычно имеет суффикс «.s» и без явного указания не создаётся. Ассемблерные файлы генерируются не в семантике Intel (принятой, например, в DOS или Windows), а в семантике AT&T.

Третья стадия. *Ассемблер* транслирует программу в так называемый объектный модуль (суффикс «.o»). Объектный файл представляет собой блоки машинного кода и данных с неопределёнными (импортируемыми) адресами ссылок на данные и процедуры в других объектных модулях и содержит список своих (определённых, экспортируемых) процедур и данных.

Четвёртая стадия. *Редактор связей (компоновщик)* связывает один или несколько объектных модулей, полученных в результате компиляции, в исполняемый файл (без суффикса) или динамически загружаемую библиотеку (суффикс «.so»). Для каждого импортируемого имени находится его определение в других модулях, упоминаемое имя заменяется его адресом (этот процесс называется «разрешением» – resolve). При этом в процессе компоновки происходит связывание программы с динамическими или статическими библиотеками (последние являются архивами объектных файлов) и добавляется код, обеспечивающий инициализацию экземпляра программы при её запуске. Если не указано иное, результирующий файл именуется a.out.

Описанная последовательность может быть начата или прервана на любой стадии, что задаётся опциями в командной строке. Например, опция *-E* указывает завершить обработку файла после стадии препроцессора (результат выдаётся на стандартный вывод); опция *-S* указывает завершить обработку файла после стадии компилятора (перед ассемблером); опция *-c* указывает завершить обработку файла перед компоновкой редактором связей. Оптимизатор **gcc** задействуется при использовании одного из вариантов опции *-O*.

Программа **gcc** определяет, какие действия предпринять с тем или иным файлом, по его суффиксу, поэтому необходимо правильно именовать файлы, в противном случае в командной строке придётся указывать дополнительные опции.

В командной строке **gcc** можно указывать несколько исходных файлов (не обязательно от одной и той же стадии), тогда в результате они будут собраны в один исполняемый файл. Это бывает удобно при разработке многомодульных приложений, когда отдельный модуль описан в отдельном исходном файле. Пример:

```
$ cat hello.c
#include <stdio.h>
main() {
       printf("hello, world\n");
}
$ gcc hello.c
$ ls –l
a.out hello.c
$ ./a.out
hello, world
```

Можно указать в командной строке ключ *-o*, чтобы для исполняемого файла задать имя, отличное от a.out.

```
$ gcc -o hello hello.c
$ ./hello
hello, world
```

Опция *-lимя* подключает соответствующую библиотеку. Например, *-lx* подключает библиотеку libx. Необходимо помнить, что включение в исходный текст программы того или иного заголовочного файла (например, math.h) недостаточно для успешной генерации исполняемого файла, поскольку заголовочный файл содержит информацию лишь для синтаксического анализатора, позволяющую компилятору сгенерировать корректный код вызова тех или иных библиотечных функций. А вот собственно интеграция с кодом библиотечных функций происходит на стадии компоновки, для которой необходим объектный код этих функций (например, файл libm.a), что достигается при помощи этого ключа (для libm – ключ *-lm)*.

Весьма полезно использовать опцию *-Wall*, которая указывает компилятору выводить все предупреждения, благодаря чему становится проще обнаружить семантические ошибки в программе.

Подробности обо всех опциях компилятора Си/Си++ можно узнать по команде **`man gcc`**, а редактора связей – по команде **`man ld`**.

7.2 Программа make

Программа **`make`** в соответствии со специальным сценарием отслеживает зависимости между заданными файлами, так что изменение одного из них влечёт выполнение определённых действий с другими файлами. Чаще всего **`make`** используется для компиляции группы программных модулей и сборки их в один проект.

Сценарий для программы **`make`** обычно хранится в файле с именем Makefile, и в этом случае можно запустить **`make`** без параметров. В противном случае имя сценария нужно передать программе **`make`** при помощи ключа *-f*:

```
make -f my_makefile
```

Рассмотрим работу с **`make`** на примере проекта baseline. Предположим, проект baseline состоит из четырёх основных модулей: main.c, create.c, update.c и delete.c. Предположим также, что файл update.c использует файл определений change.h. Взаимосвязи файлов схематично представлены на рисунке 7.6.

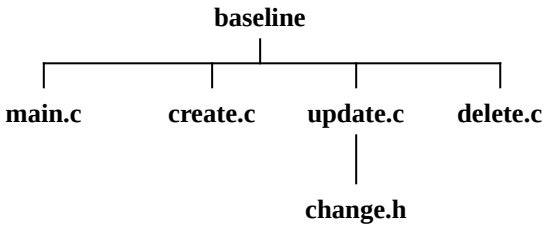

Рис. 7.6. Пример взаимосвязи файлов (проект baseline)

Для сборки данного проекта можно использовать возможность **`gcc`** указывать в командной строке несколько исходных файлов, тогда, указав все «.c»-файлы проекта, мы получим исполняемый файл, содержащий все указанные программные модули. Однако при изменении любого из модулей при таком способе сборки придётся перекомпилировать все модули.

Более эффективное решение предоставляет программа **`make`**. Можно компилировать перечисленные модули по отдельности, доходя до стадии сборки, но не вызывая редактор связей, а когда получены объектные файлы всех модулей, собрать их воедино при помощи редактора связей. Сценарий Makefile, описывающий необходимые взаимосвязи, для данного случая выглядит следующим образом:

```
# Makefile для построения программы baseline
CFLAGS=-g -Wall

baseline: main.o create.o update.o delete.o
    gcc -o baseline main.o create.o update.o delete.o

main.o: main.c
```

```
create.o: create.c
update.o: update.c change.h
delete.o: delete.c
```

Строки, начинающиеся с октоторпа (#), являются комментариями и игнорируются. Далее следуют определения некоторых переменных окружения. В данном примере объявлена переменная CFLAGS, которая хранит опции компилятора по умолчанию. Сценарий содержит несколько блоков вида:

```
<цель>: <зависимости>
      <команда>
      <команда>
      ...
```

В данном примере результат компилирования проекта (первая цель в сценарии Makefile) – файл baseline. Файл зависит от объектных модулей main.o, create.o, update.o и delete.o. Чтобы получить этот файл, должна быть выполнена команда, указанная в этом блоке. Команды в блоке записываются не в начале блока, а после символа табуляции (важно, чтобы это был именно символ табуляции, а не пробелы). Остальные цели описывают зависимости этих объектных модулей. Например, файл main.o зависит от файла main.c. Если для какой-то цели не указаны команды, как получить целевой файл из его зависимостей, срабатывает механизм умолчаний. Так, файлы с суффиксом «.c» обрабатываются Си-компилятором; файлы с суффиксом «.cpp» обрабатываются Си++-компилятором и т. п. Таким образом, программа **make** определяет, что для получения перечисленных объектных модулей надо скомпилировать соответствующие файлы.

```
$ make
gcc -c main.c
gcc -c create.c
gcc -c update.c
gcc -c delete.c
gcc -o baseline main.o create.o update.o delete.o
```

Если теперь внести изменения в файл change.h, то при запуске программа **make** по времени модификации определит, что он новее, чем зависящий от него файл update.o и перекомпилирует этот объектный модуль. После этого в соответствии со сценарием потребуется перекомпиляция основной цели – файла baseline, тогда как остальные объектные модули останутся неизменными.

```
$ make
gcc -O -c update.c
gcc -o baseline main.o create.o update.o delete.o
```

Если выполнить make в то время, когда не было никаких изменений, то она выводит предупреждение:

```
$ make
'baseline' is up to date
```

Таким образом, программа **make** значительно облегчает компиляцию сложных многомодульных проектов. Более того, набор утилит GNU содержит также средства для автоматической генерации файлов сценариев программы **make** при компиляции проекта на разных платформах. При помощи утилит **autoconf** и **automake** генерируется специальный сценарий для командного интерпретатора (обычно этот сценарий называется **configure**). Это сценарий распространяется вместе с исходным текстом проекта и при запуске определяет тип платформы и необходимые для компиляции проекта средства. В результате своей работы такой сценарий создаёт Makefile с необходимыми опциями для компилятора, редактора связей и т. п., а также заголовочный файл (обычно config.h), в котором указываются директивы препроцессора, управляющие особенностями компиляции на данной платформе.

7.3 Отладчик

Для отладки программ в GNU/Linux предназначена программа GNU Debugger (**gdb**), которая предоставляет средства выполнения программы по шагам, просмотра переменных программы, регистров процессора, стека и т. п. Программа имеет командный интерфейс – при запуске выдаётся приглашение, после которого вводятся команды **gdb** и их параметры. Отладчик позволяет пошагово выполнять программу, расставлять в ней контрольные точки, наблюдать значения переменных, отслеживать стек вызовов процедур и проч.

В GNU/Linux при аварийном завершении программы может быть сформирован файл образа памяти программы в момент аварии. Этот файл можно использовать в отладчике, чтобы найти место аварийного завершения программы и определить вероятную причину.

Отладочная информация – это дополнительные данные, добавляемые в исполняемый файл, представляющие собой информацию об именах переменных и функций, номера строк исходного текста программы, позволяющие связать машинные инструкции с операторами языка высокого уровня. Отладочная информация су-

щественно увеличивает размер получаемого исполняемого файла и отменяет некоторые способы оптимизации, поэтому по умолчанию отладочная информация в исполняемый файл не добавляется. Однако эффективная работа с отладчиком без неё невозможна. Поэтому на стадии разработки и тестирования программисты обычно включают её в исполняемый файл, а финальная версия программы для эксплуатации широким кругом пользователей уже формируется без отладочной информации. **`Gcc`** включает отладочную информацию в исполняемый файл, если указана опция *-g*.

Некоторые команды отладчика:

- **`help`** – помощь,
- **`run`** – запустить программу,
- **`print`** – вывести значение выражения (можно использовать переменные программы),
- **`backtrace`** – вывести стек вызовов процедур,
- **`list`** – вывод фрагмента листинга программы,
- **`start`** – загрузить программу и начать пошаговое выполнение,
- **`step`** – выполнить очередной шаг в программе,
- **`break`** – установить контрольную точку по коду,
- **`watch`** – установить контрольную точку по данным,
- **`continue`** – продолжить выполнение программы.

Контрольные вопросы и задания

1. Компиляция готового программного продукта.
   1) С указанного преподавателем ресурса загрузите два каких-либо архива с исходными текстами (вместо астериска в имени файла указывается версия): bc*.tar.bz2, file*.tar.bz2, links*.tar.bz2, mp3info*.tar.bz2, wget*.tar.bz2, recode*.tar.bz2, units*.tar.bz2, stat*.tar.bz2 или др.
   2) Распакуйте архив в своём домашнем каталоге (воспользуйтесь командой **`tar`**).
   3) Изучите порядок компиляции и установки программного продукта (внутри архива найдите файлы README, INSTALL и т. п.). Обычно это включает выполнение следующих действий:

```
./configure --help
./configure <необходимые параметры>
make
make install
```

   4) Последняя команда должна выполняться от имени привилегированного пользователя, если программный продукт устанавливается в общесистемный каталог (/usr или /usr/local). Вам же надо скомпилировать и установить программный продукт в своём домашнем каталоге, в подкаталоге soft (~/soft).
   5) Убедитесь в работоспособности программного продукта.
2. Изучение отладчика **`gdb`**.
   1) Скомпилируйте программу с нарушением доступа к памяти. Например:

```
#include <stdio.h>
int main() {
 float f;
 scanf("%f",f);
 printf("f=%f",f);
 return 0;
}
```

   2) Запустите её. Если создание образа памяти (core dump) отключено, включите его при помощи команды **`ulimit`**.
   3) Откройте программу и её дамп памяти в отладчике.

```
gdb ./a.out ./core
```

   4) Проследите стек вызовов функций (**`backtrace`**).
   5) Перекомпилируйте программу с поддержкой отладочной информации и сравните результат.