Тема №4. Компилятор GCC. Оптимизация

Знакомство с компилятором GCC

Средством, традиционно используемым для создания программ для ОС Linux, является инструментарий разработчика GNU. Проект GNU был основан в 1984 году Ричардом Столлманом. Его необходимость была вызвана тем, что в то время сотрудничество между программистами было затруднено, так как владельцы коммерческого программного обеспечения чинили многочисленные препятствия такому сотрудничеству. Целью проекта GNU было создание комплекта программного обеспечения под единой лицензией, которая не допускала бы возможности присваивания кем-то эксклюзивных прав на это ПО. Частью этого комплекта и является инструментарий разработчика, которым мы будем пользоваться, и который входить во все дистрибутивы Linux.

Одним из этих инструментов является компилятор GCC. Первоначально эта аббревиатура расшифровывалась, как GNU C Compiler. Сейчас она означает – GNU Compiler Collection.

Компиляция простой программы

Создадим нашу первую программу с помощью GCC.

Создайте отдельный каталог `hello`. Это будет каталог нашего первого проекта. В нём создайте текстовый файл `hello.c` со следующим текстом:

```
#include <stdio.h>

int main(void) {
     printf("Hello world!\n");
     return(0);
}
```

Затем в консоли зайдите в каталог проекта. Наберите следующую команду:

$ gcc hello.c

В каталоге появился новый файл `a.out`. Это исполняемый файл. Запустим его. Наберите в консоли:

$ ./a.out

Hello world!

Компилятор gcc, по умолчанию, присваивает всем созданным исполняемым файлам имя `a.out`. Если хотите назвать его по-другому, нужно задать компилятору ключ `-o` и имя, которым вы хотите его назвать. Ключ `-o` является лишь одним из многочисленных ключей компилятора gcc. Давайте выполним такую команду:

$ gcc hello.c -o hello

Мы видим, что в каталоге появился исполняемый файл с названием `hello`. Запустим его:

$ ./hello

Как видите, получился исполняемый файл с заданным нами именем.

Вы, конечно, обратили внимание, что, когда мы запускаем программу из нашего каталога разработки, мы перед названием файла `./`. Зачем же мы это делаем?

Дело в том, что, если мы наберём только название исполняемого файла, операционная система будет искать его в каталогах `/usr/bin` и `/usr/local/bin`, и, естественно, не найдёт. Каталоги `/usr/bin` и `/usr/local/bin` – системные каталоги размещения исполняемых программ. Первый из них предназначен для размещения стабильных версий программ, как правило, входящих в дистрибутив Linux. Второй – для программ, устанавливаемых самим пользователем. По умолчанию при сборке программы устанавливаются в каталог `/usr/local/bin`. Крайне нежелательно помещать что-либо лишнее в `/usr/bin` или удалять что-то оттуда вручную, в виду возможного краха системы. Там должны размещаться программы, за стабильность которых отвечают разработчики дистрибутива.

Чтобы запустить программу, находящуюся в другом месте, надо прописать полный путь к ней, например так:

/home/user/hello/hello

Или другой вариант: прописать путь относительно текущего каталога, в котором вы в данной момент находитесь в консоли. При этом одна точка означает текущий каталог, две точки – родительский. Например, команда `./hello` запускает программу `hello`, находящуюся в текущем каталоге, команда `../hello` – программу `hello`, находящуюся в родительском каталоге, команда `./hello/hello` – программу во вложенном каталоге, находящимся внутри текущего.

Есть возможность добавлять в список системных путей к программам дополнительные каталоги. Для этого надо добавить новый путь в системную переменную `PATH`. Сейчас мы не будем заострять на этом внимание. Переменные окружения – это отдельный разговор.

Выявление ошибок

Предупреждения компилятора — неоценимое подспорье разработчику. По умолчанию, предупреждения в GCC отключены. Посмотрим к чему это может привести:

```
#include <stdio.h>

int main(void) {
        printf("Two plus two is %f\n", 4);
        return(0);
}
```

Проведя компиляцию и запустив программу мы получим следующий результат:

$gcc bad.c -o bad

$./bad

Two plus two is 2.585495

Полученный результат не верен. В чем может быть ошибка? С первого взгляда ошибку можно и не заметить, уж больно она коварная. Для вывода целого числа 4 задана маска `%f`, отвечающая за вывод чисел с плавающей точкой.

Чтобы компилятор показал нам подобную западню необходимо при компиляции задать ключ `-Wall`. Данная опция включает режим отображения замеченных

компилятором ошибок.

$gcc -Wall bad.c -o bad

bad.c: In function 'main':

bad.c:6: warning: double format, different

type arg (arg 2)

Основные этапы работы компилятора GCC

Процесс создания программы в GCC разбит на три основных этапа: обработка препроцессором, компиляция и компоновка.

Препроцессор подставляет в основной файл содержимое всех заголовочных файлов, указанных в директивах `#include`. В заголовочных файлах обычно находятся объявления функций, используемых в программе, но не определённых в тексте программы. Их определения находятся где-то в другом месте: или в других файлах с исходным кодом или в двоичных библиотеках.

Вторая стадия – компиляция. Она заключается в превращении текста программы на языке C/C++ в набор машинных команд. Результат сохраняется в объектном файле. Разумеется, на машинах с разной архитектурой процессора двоичные файлы получаются в разных форматах, и на одной машине невозможно запустить двоичный файл, собранный на другой машине (разве только, если у них одинаковая архитектура процессора и одинаковые операционные системы). Вот почему программы для UNIX-подобных систем распространяются в виде исходных кодов: они должны быть доступны всем пользователям, независимо от того, типа используемого процессора и операционной системы.

Последняя стадия – компоновка. Она заключается в связывании всех объектных файлов проекта в один, связывании вызовов функций с их определениями, и присоединением библиотечных файлов, содержащих функции, которые вызываются, но не определены в проекте. В результате формируется исполняемый файл. Если какая-то функция в программе используется, но компоновщик не найдёт место, где эта функция определена, он выдаст сообщение об ошибке, и откажется создавать исполняемый файл.

Теперь рассмотрим на практике, как всё это выглядит. Напишем простейший калькулятор, способный складывать, вычитать, умножать и делить. При запуске он будет запрашивать по очереди два числа, над которыми следует произвести действие, а затем потребует ввести знак арифметического действия. Это могут быть четыре знака: «+», «–», «*», «/». После этого программа выводит результат и завершается.

Создадим для проекта новую папку `calc`, в ней создадим файл `calc.c`.

```
#include <stdio.h>

int main(void) {
float num1;
float num2;
char op;
```

```
printf("First number: ");
scanf("%f",&num1);
printf("Second number: ");
scanf("%f",&num2);
printf("Оператор ( + - * / ): ");

while ((op = getchar()) != EOF) {
     if (op == '+') {
           printf("%6.2f\n",num1 + num2);
           break;
     } else if(op == '-') {
           printf("%6.2f\n",num1 - num2);
           break;
     } else if(op == '*')      {
           printf("%6.2f\n",num1 * num2);
           break;
     } else if(op == '/') {
           if(num2 == 0) {
                 printf("Error: zero divide!\n");
                 break;
           } else {
                 printf("%6.2f\n",num1 / num2);
                 break;
           }
     }
}
return 0;
}
```

Итак, первым делом, как было сказано, выполняется препроцессирование. Для того, чтобы посмотреть, что на этом этапе делается, воспользуемся ключом `-E`. Этот ключ останавливает выполнение программы на этапе обработки препроцессором. В результате получается файл исходного кода с включённым в него содержимым заголовочных файлов.

В нашем случае мы включали один заголовочный файл – `stdio.h` – библиотеку стандартных функций ввода-вывода.

Введите следующую команду:

$ gcc -E calc.c -o temp.cpp

Полученному файлу мы дали имя `temp.cpp`. Откройте его. Обратите внимание на то, что он весьма длинный. Это потому что в него вошёл весь код заголовочного файла `stdio.h`. Кроме того, препроцессор добавил некоторые теги, указывающие компилятору способ связи с объявленными функциями. Основной текст нашей программы находится в самом низу получившегося файла.

Посмотрим теперь следующий этап. Создадим объектный файл. Объектный файл представляет собой трансляцию исходного кода в двоичный код, без связи вызываемых функций с их определениями. Для формирования объектного файла служит опция `-c`.

$ gcc -c calc.c

Название получаемого файла можно не указывать, так как компилятор просто берёт название исходного и меняет расширение `.c` на `.o` (указать можно, если нам захочется назвать его по-другому).

Если мы создаём объектный файл из исходного файла, уже обработанного препроцессором (например, такого, что мы получили выше), то мы должны обязательно явно указать, что компилируемый файл является файлом исходного кода, обработанный препроцессором, и имеющий теги препроцессора. В противном случае он будет обрабатываться, как обычный файл C++, без учёта тегов препроцессора, а значит связь с объявленными функциями не будет устанавливаться. Для явного указания на язык и формат обрабатываемого файла служит ключ `-x`. Файл C++, обработанный препроцессором обозначается `cpp-output`.

$ gcc -x cpp-output -c calc.cpp

Наконец, последний этап – компоновка. Получаем из объектного файла исполняемый.

$ gcc calc.o -o calc

Можно его запускать.

$ ./calc

Зачем нужна возня с промежуточными этапами? Не лучше ли просто один раз выполнить `gcc calc.c -o calc`?»

Дело в том, что настоящие программы очень редко состоят из одного файла. Как правило исходных файлов несколько, и они объединены в проект. И в некоторых исключительных случаях программу приходится компоновать из нескольких частей, написанных на разных языка. В этом случае приходится запускать компиляторы разных языков, чтобы каждый получил объектный файл из своего исходного кода, а затем уже эти полученные объектные файлы компоновать в исполняемую программу.

Сборка программы из нескольких исходных файлов

Обычно простые программы состоят из одного исходного файла. Дело обстоит несколько сложнее, если приходиться иметь дело с крупным проектом. При работе с такой программой может возникнуть несколько достаточно серьезных проблем:

- Файл, становясь большим, увеличивает время компиляции, и малейшие изменения в исходном тексте автоматически вынуждают тратить время программиста на перекомпиляцию программы.

- Если над программой работает много человек, то практически невозможно отследить сделанные изменения.

- Процесс правки и само ориентирование при большом исходном тексте становится сложным и поиск небольшой ошибки может повлечь за собой вынужденное "изучение" кода заново.

Это далеко не все проблемы, которые могут в рассматриваемом случае. Поэтому при разработке программ рекомендуется разделять их на куски, которые

функционально ограничены и закончены.

Для того, чтобы вынести функцию или переменную в отдельный файл надо перед ней поставить зарезервированное слово `extern`. Давайте для примера создадим программу из нескольких файлов. Сначала создадим главную программу, в которой будут две внешние процедуры. Назовем этот файл `main.c`:

```
#include <stdio.h>

extern int f1();
extern int f2();

int main() {
	int n1, n2;

	n1 = f1();
	n2 = f2();

	printf("f1() = %d\n",n1);
	printf("f2() = %d\n",n2);

	return 0;
}
```

Теперь создаем два файла, каждый из которых будет содержать полное определение внешней функции из главной программы. Файлы назовем `f1.c` и `f2.c`:

```
// файл f1.c
int f1() {
	return 2;
}

// файл f2.c
int f2()  {
	return 10;
}
```

Компилировать можно все файлы одновременно одной командой, перечисляя составные файлы через пробел после ключа `-c`:

$ gcc -c main.c f1.c f2.c

Или каждый файл в отдельности:

$ gcc -c f1.c
$ gcc -c f2.c
$ gcc -c main.c

В результате работы компилятора мы получим три отдельных объектных файла: `main.o`, `f1.o`, `f2.o`.

Чтобы их собрать в один файл с помощью `gcc` надо использовать ключ `-o`, при этом компоновщик соберет все файлы в один:

$ gcc main.o f1.o f2.o -o result

В результате вызова полученной программы `result` командой:

$ ./result
f1() = 2
f2() = 10

Теперь, мы изменим какую-то из процедур, например `f1()`:

```
int f1() {
        return 25;
}
```

Компилировать заново все файлы не придется, а понадобится лишь скомпилировать измененный файл и собрать результирующий файл из частей:

$ gcc -c f1.c
$ gcc main.o f1.o f2.o -o result
$ ./result
f1() = 25
f2() = 10

Таким образом можно создавать большие проекты, которые не будут отнимать много времени на компиляцию и поиск ошибок.

Оптимизация

В целях контроля времени компиляции и количества используемой памяти, а также в целях выбора разумного компромисса между скоростью выполнения и объемом конечного исполняемого файла, GCC предоставляет ряд уровней оптимизации(O1, O2, O3, O4). В дополнение к перечисленным уровням оптимизации имеются специлизированные методы оптимизации. Уровень оптимизации выбирается заданием соответсвующего ключа `-O`*`LEVEL`*, где *`LEVEL`* это число от 0 до 3. Эффекты от применения различных уровней оптимизации описаны ниже:

-O0 или отсутствие ключа опимизации (по умолчанию)

На этом уровне оптимизации GCC не производит оптимизации и компилирует исходный код наиболее простым путем. Каждая команда исходного кода преобразуется непосредственно в соответсвующую инструкцию в исполняемом файле, без каких-либо существеннх изменений. Данный режим подходит для этапа отладки программы или в случае отсутвия необходимости в оптимизации.

-O1 или -O

Данный уровень оптимизации использует наиболее общие методы оптимизации, не влекущие за собой необходимости в поиске компромисса между скоростью выполения и объемом исполняемого файла. Итоговый исполняемый файл будет меньше чем при использовании ключа `-O0`, а скорость выполенения программы будет выше. Более дорогостоящие виды оптимизации, такие как планирование инструкций, на данном уровне не

производятся. Компиляция с использованием ключа `-O1` може занимать меньше времени, чем компиляция с клюем `-O0`, в следствии уменьшения количества обрабатываемой информации после выполения простых этапов оптимизации.

-O2

Данная опция помимо методов оптимизации используемых на предыдущем уровне включает ряд дополнительных механизмов оптимизации, в частности планирование инструкции. Используются лишь методы оптимизации не требующие поиска компромисса межу скоростью исполнения программы и размером исполняемого файла, в следствие чего исполняемый файл не должен увеличиться в размерах. Компилятору требуется боольше времени и памяти для компиляции программ, чем при использовании ключа `-O1`. Данная уровень оптимизации предпочтителен при развертывании программ, так как обеспечивае максимально возможную оптимизацию без необходимости увеличивать размер исполянемого файла. Этот метод используется, как метод по умолчанию для сборки программ и библиотек GNU.

-O3

Данный ключ включает более дорогостоящие методы оптимизации, такие как подстановка функции[1], в дополнение ко всем методам применяемым на уровнях `-O2` и `-O1`. Уровень `-O3` может увеличить скорость выполения программы, но точно также может увеличить размер исполняемого файла. Иногда, данный вид оптимизации может приводить к понижениею производительности программы.

-funroll-loops

Данный ключ отвечает за включение механизма развертывания циклов[2]. Не зависит от других опций оптимизации. При развертывании циклов объём исполняемого файла значительно увеличивается. Часто при значительном увеличении объема исполняемого файл прироста в скорости получить не удается, поэтому к каждом конкретном случае требуется дополнительное исследование.

-Os

Данный ключ отвечает за методы оптимизации приводящие к уменьшению размера исполняемого файла. Применяется с целью создания максимально компактного исполняемого файла, в первую очередь для систем с ограничением на оперативную память или дисковое пространосвто. В некотороых случаях более компакнтый исполняемый файл будет выполнятся быстрее, в следствие более оптимальной работы с кэшем.

Важно помнить, что преимущества высоких уровней оптимизации могут быть невилирована их стоимостью: сложностью при отладке, высокими затратами времени и памяти при компиляции. Для большинства проектов разумно использовать оптимизацию `-O0` для отладки и `-O2` для сборки готового проекта.

---

1 Вставка тела небольших функций/методов непосредственно в место их вызова, уменьшая затраты на их вызов

2 Превращение цикла в линейную последовательность команд

Примеры

Рассмотрим пример демонстрирующий эффекты от применения различные уровней оптимизации:

```
#include <stdio.h>

double powern (double d, unsigned n) {
  double x = 1.0;
  unsigned j;

  for (j = 1; j <= n; j++)
    x *= d;

  return x;
}

int main (void) {
  double sum = 0.0;
  unsigned i;

  for (i = 1; i <= 100000000; i++)
    {
      sum += powern (i, i % 5);
    }

  printf ("sum = %g\n", sum);
  return 0;
}
```

Функция main содежит цикл, вызывающий функцию powern. Данная функция вычисляет n-ую степень числа с плавающей точкой, выполняя последовательное перемножение. Функция powern была выбрана с целью демонстрации механизмов развертывания циклов и подстановки функций. Время выполнения программы может быть вычислено путем использования команды time, входящей в состав оболочки GNU Bash.

Вот результаты приведенной выше программы скомпилированой с использованием процессора 566MHz Intel Celeron с 16KB кэша L1 и 128KB кэша L2. Использовался компилятор GCC 3.3.1 под ОС GNU/Linux:

```
$ gcc -Wall -O0 test.c -lm
$ time ./a.out
real    0m13.388s - общее время выполнение
user    0m13.370s - чистое процессорное время
sys     0m0.010s  - время ожидание отклика системы

$ gcc -Wall -O1 test.c -lm
$ time ./a.out
```

```
real    0m10.030s
user    0m10.030s
sys     0m0.000s

$ gcc -Wall -O2 test.c -lm
$ time ./a.out
real    0m8.388s
user    0m8.380s
sys     0m0.000s

$ gcc -Wall -O3 test.c -lm
$ time ./a.out
real    0m6.742s
user    0m6.730s
sys     0m0.000s

$ gcc -Wall -O3 -funroll-loops test.c -lm
$ time ./a.out
real    0m5.412s
user    0m5.390s
sys     0m0.000s
```

Из результата видно что оптимизация уровней `-O1`, `-O2` и `-O3` дает прирост производительности, сопоставимый с производительностьюр программы без оптимизации. Добавление ключа `-funroll-loops` дает дальнейший прирост производительности. Скорость программы на максимальном уровне оптимизации увеличивается в двое по сравнению с производительностью без оптимизации.

Обратите внимание, что для столь небольшой программы могут быть значительные различия на разных системах и весриях компилятора. К примеру, на мобильной системе с процессором 2.0GHz Intel Pentium 4M результат от различных уровней сопоставим с приведенным выше, за исключением оптимизации уровня `-O2`, где производительность просела по сравнению с уровнем `-O1`. Это иллюстрирует важное замечание: оптимизация не обязательно увеличит производительность программы.

Задание

1. Выполнить продемонстрированные выше примеры.
2. На основе примера, демонстрирующего различные уровни оптимизации, написать сценарий, выполняющий следующие действия:
    - Вычисление времени выполнения программы.
    - Вычисление занимаемого исполняемым файлом дискового пространства(в байтах).

    Сценарий должен принимать следующие параметры командной строки:

    - Имя исходного файла программы.
    - Требуемые ключи оптимизации, если таковые нужны.

    Вывод программы должен содержать следующую информацию:

    - Текущие ключи оптимизации
    - Время затраченное программой на выполнение
    - Занимаемое программой дисковое пространство