Сборка программы на языке C

А. Г. Фенстер, fenster@fenster.name

9 февраля 2010 г.

1 Сборка программы из нескольких файлов

Программы на языке C обычно состоят из нескольких файлов с исходным кодом. Рассмотрим этапы сборки программы в исполняемый файл. Примеры будут приведены для ОС Linux и компилятора gcc. При использовании Microsoft Visual Studio всё происходит так же (это можно наблюдать в build log проекта).

1.1 Этапы сборки программы

1. Раздельная компиляция каждого файла с исходным кодом (`.c`), которая включает в себя

   (a) препроцессинг — обработка директив *препроцессора* `#define`, `#include` и прочих, результатом которой будет код на языке C уже без этих директив; о работе препроцессора мы поговорим на отдельном занятии;

   (b) компиляцию полученного текста программы на C в объектный файл (`.o` / `.obj`).

   ```
   cc -c file1.c
   ...
   cc -c fileN.c
   ```

2. Компоновка (сборка, линковка) итогового *исполняемого файла*:
   `cc` **`-o executable`** `file1.o ... fileN.o`

Рассмотрим простую программу из двух файлов: функция `f`, объявленная в первом файле, использует глобальную переменную `a` из второго файла; функция `main` во втором файле использует функцию `f`.

`file1.c`

```
int f(void)
{
    return (a + 1);
}
```

`file2.c`

```
#include <stdio.h>
int a = 0;
int main()
{
    printf("%d\n", f());
    return 0;
}
```

При попытке откомпилировать файл `file1.c` мы получим сообщение об ошибке (неизвестный идентификатор `a`), при попытке откомпилировать файл `file2.c` получим сообщение о том, что функция `f` неизвестна (правда, чтобы заставить `gcc` считать это ошибкой, нужно указать дополнительные ключи компиляции `-Wall -Werror`).

```
$ gcc -c file1.c
file1.c: In function 'f':
file1.c:3: error: 'a' undeclared (first use in this function)
file1.c:3: error: (Each undeclared identifier is reported only once
file1.c:3: error: for each function it appears in.)

$ gcc -Wall -Werror -c file2.c
cc1: warnings being treated as errors
file2.c: In function 'main':
file2.c:5: error: implicit declaration of function 'f'
```

Чтобы каждый из файлов мог быть откомпилирован, необходимо описать (declare, не define) используемые в нём переменные и функции, определённые (defined) в других файлах. Добавленные строки отмечены знаком `/*!*/`:

`file1.c`

```
/*!*/ extern int a;
int f(void)
{
    return (a + 1);
}
```

`file2.c`

```
#include <stdio.h>
/*!*/ int f(void);
int a = 0;
int main()
{
    printf("%d\n", f());
    return 0;
}
```

Ключевое слово `extern` можно читать как «определена где-то в другом месте»: мы сообщаем компилятору, что переменная `a` будет существовать на этапе компоновки программы, потому что определена в дру-

гом файле. Аналогично во втором файле утверждается, что где-то будет определена функция `f`.

> **Важно:** не путайте слова «определение» (definition) и «объявление» (declaration). Мы будем говорить, что некоторый объект *определяется* (is defined) в данном фрагменте кода, если фрагмент кода требует физически разместить его в памяти, и *объявляется* (is declared), если его физическое размещение в памяти происходит при обработке какого-либо другого фрагмента кода.

Сейчас раздельная компиляция файлов `file1.c` и `file2.c` пройдёт успешно:

```
$ gcc -c file1.c
$ gcc -c file2.c
$ ls
file1.c  file1.o  file2.c  file2.o
```

Каждый из полученных объектных файлов (`file1.o` и `file2.o` для линукса, а в Windows обычно `.obj`) по отдельности не содержит всего необходимого для того, чтобы программа заработала — забыв один из них при компоновке, получим ошибку:

```
$ gcc -o executable file2.o
file2.o: In function ‘main’:
file2.c:(.text+0xa): undefined reference to ‘f’
collect2: ld returned 1 exit status
```

Оба объектных файла вместе можно скомпоновать (link) в исполняемый файл и запустить его:

```
$ gcc -o executable file1.o file2.o
$ ./executable
1
```

1.2 Типы ошибок, выдаваемых компилятором

Из сказанного выше можно сделать вывод, что ошибки, выдаваемые компилятором, обычно принадлежат к одному из двух типов:

1. Ошибки компиляции (лексическая, синтаксическая ошибка — на одном из следующих занятий мы разберём их подробнее); обычно указано место в файле с исходным кодом, в котором встретилась ошибка;

2. Ошибки компоновки (неопределённая переменная или функция); возникает, когда файлы программы прошли компиляцию, но на этапе сборки одного общего исполняемого файла выясняется, что какая-либо функция или переменная так и осталась неопределённой. В этом случае указать точное место в коде, вызвавшее ошибку, обычно затруднительно.

1.3 Использование заголовочных файлов

Если бы при использовании любых переменных, функций или типов из других файлов необходимо было бы перечислять их все в своём файле, писать программы было бы весьма неудобно (помните ли вы наизусть типы аргументов и возвращаемых значений всех используемых вами стандартных функций?) — поэтому разработчики библиотек функций выделяют их описания в отдельные файлы с расширением `.h` (от header — заголовок). Например, вместе с `file1.c` поставлялся бы `file1.h` со следующим содержимым:

```
int f(void);
```

Аналогично, `file2.h` содержал бы

```
extern int a;
```

В таком случае в `file1.c` достаточно было бы написать `#include "file2.h"` (и наоборот). Это намного удобнее явного перечисления необходимых функций и переменных.

В заголовочном файле могут присутствовать только `extern`-описания переменных, заголовки функций (без реализации), определения структур и `typedef`-определения типов.

1.4 Include guards

Если заголовочный файл содержит определения структур или типов (никакие другие определения в нём писать не следует — только объявления), может возникнуть *проблема двойного подключения*: если каким-то образом текст заголовочного файла будет включён в файл `.c` дважды, произойдёт ошибка — нарушение «правила одного определения», запрещающего определять один и тот же объект более одного раза. Пример:

list.h	hash.h

`struct item`
`{`
`  int data;`
`  struct item *next;`
`};`
 	
`...`
	`#include "list.h"`
	
	`...`

Если автор файла `file.c` включит в него при помощи `#include` одновременно и `list.h`, и `hash.h`, после препроцессинга в `file.c` окажется два одинаковых определения структуры `struct item`, что приведёт к ошибке компиляции. Решить проблему можно заключением всего содержимого заголовочных файлов в директивы `#ifndef ... #endif`, называемые (в данном случае) include guards:

list.h	hash.h

`#ifndef __LIST_H`
`#define __LIST_H`
`struct item`
`{`
`  int data;`
`  struct item *next;`
`};`
 	
`...`
`#endif`
	`#ifndef __HASH_H`
	`#define __HASH_H`
	`#include "list.h"`
	
	`...`
	`#endif`

При первом включении `list.h` символ `__LIST_H` станет определённым, и при попытке включить этот же файл во второй раз директива `#ifndef` (if not defined) не позволит вставить этот же код ещё раз.

Современные компиляторы позволяют вместо include guards писать специальную (нестандартную) директиву `#pragma once`.

Подробно директивы препроцессора мы рассмотрим на отдельном занятии.

2 Классы памяти

Рассмотрим различные классы памяти (storage classes — возможно, лучше перевести как «классы хранения»), соответствующие различным переменным в программе на C.

2.1 Класс памяти auto: локальные переменные

Переменные, определённые внутри функций или блоков, называют *локальными* или *автоматическими* (поскольку память под них выделяется автоматически). Говорят, что такие переменные соответствуют классу памяти `auto` (и это даже можно явно указать в коде: два фрагмента ниже эквивалентны).

```
void f(void)
{
    int a;
    ....
}
```

```
void f(void)
{
    auto int a;
    ....
}
```

Если переменной не присвоено значение при определении (как в примере выше), её значение будет **не определено** (в ней будет содержаться «мусор» из памяти).

Автоматические переменные теряют своё значение при выходе из блока, в котором они были определены.

2.2 Локальные переменные с модификатором `static`

Если **локальная** переменная (переменная внутри функции) определена с модификатором `static` (например, `static int a;`), то она будет сохранять своё значение между вызовами функции. Инициализация такой переменной сработает только при первом вызове функции, а если переменной не присвоено значение при её определении, её значением будет ноль.

В следующем примере `counter` считает, сколько раз была вызвана функция `f`:

```
void f(void)
{
    static int counter;
    counter++;
}
```

2.3 Глобальные переменные

Глобальные переменные определяются вне функций. Они существуют в памяти на протяжении всей работы программы. Если глобальной переменной не присвоено значение при её определении, она будет равна нулю.

Чтобы глобальная переменная была «видна» из другого файла (другой *единицы компиляции*), в другом файле она должна быть объявлена как `extern`.

2.4 Глобальные переменные с модификатором `static`

Если **глобальная** переменная (переменная вне функций) определена с модификатором `static`, она будет доступна только из того файла, в котором она определена. Из других файлов получить доступ к ней не получится даже с использованием `extern`.

Такие переменные удобно использовать для хранения данных, общих для функций из одного файла, но не предназначенных для использования другими функциями. Например, реализация стека на связном списке (функции `push`, `pop`, `empty`) могут обращаться к первому элементу списка (`head`), но тем, кто использует стек, эта переменная не нужна — следовательно, её можно определить с модификатором `static`.

Модификатор `static` также можно использовать при определении функций; такие функции тоже будут доступны только из того файла, в котором они были определены.

2.5 Регистровые переменные

Указав для **локальной** переменной модификатор `register` (например, `register int a;`), мы советуем компилятору разместить её в регистре процессора (а не в оперативной памяти), что ускорит обращение к ней. Регистров ограниченное число, и компилятор вправе как прислушаться к нашему совету, так и проигнорировать его. Современные компиляторы обычно принимают решение о размещении переменной в регистре самостоятельно. Тем не менее, независимо от того, размещена ли реально переменная в регистре или нет, к переменной `register` нельзя применять оператор взятия адреса `&`.

3 Вызов функции

В языке C вызов функции происходит *по значению*. Это означает, что если определена функция

```
int f(int a)
```

```
{
    return (a + 1);
}
```

и она вызывается из программы так:

```
x = f(7);
```

то значение 7 будет скопировано в область памяти, соответствующую переменной `a`, а затем результат функции будет скопирован в переменную `x`. Параметр `a` является локальной переменной для функции `f`, начальное значение которого определяется вызовом функции; функция вольна делать с ним что угодно:

```
void g(int a)
{
    a++;
}

...
g(6);

...
int x = 5;
g(x);
/* здесь x == 5 */
```

Изменяя переменную `a`, функция меняет лишь свою копию данного ей значения, что никак не может отразиться ни на переменной `x` во втором вызове, ни тем более на константе 6.

Другое дело — языки, в которых параметры передаются *по ссылке*, например, Perl. В нём аналогичная программа будет работать иначе:

```
sub f
{
    $_[0]++; # так в перле обращаются к первому параметру функции
}

f(6);
```

Исполнение этой программы выдаст ошибку: `Modification of a read-only value attempted at test.pl line 3.`

В некоторых языках, например, Pascal и C++, можно указывать для конкретного параметра, как передавать его: по ссылке или по значению. В C++ понятие ссылки расширено ещё сильнее: функция может *вернуть ссылку*, тогда вполне допустимым будет код

```
int a;

int& f(void) // C++: int& - ссылка на int
{
    return a;
}

...
f() = 1; // изменяется переменная a
```

и такая возможность реально используется, например, при перегрузке оператора `[]`. Но об этом вам будут рассказывать только на втором курсе.

Для языка C правило формулируется просто: **изменение формальных параметров функции не приводит к изменению значений переданных ей фактически параметров в том фрагменте кода, откуда эта функция была вызвана.** Если же мы хотим это правило нарушить (например, сделать функцию `swap`, меняющую местами значения своих параметров), необходимо пользоваться указателями и передавать в функцию *адреса* параметров для изменения — об этом будет следующее занятие.

Копирование, происходящее при возвращении результата из функции, не менее важно. Пока результат имеет простой тип (например, `int`), проблем не возникает. Но вот вернуть из функции локальный массив уже невозможно: массив — это адрес его начального элемента в памяти, локальный массив имеет класс памяти `auto` и уничтожается при выходе из функции, и если вернуть его в `return` — будет warning компилятора `warning: function returns address of local variable`, а в том месте, где значение принимается, мы получим адрес места в памяти, где **когда-то был** массив.

Если же мы пометим тот локальный массив как `static`, возвращать его мы сможем смело: память, выделенная под статические переменные, существует до окончания работы программы.

```
int *f(void)
{
    int A[5] = { 1, 2, 3, 4, 5 };
    /* корректно вернуть A невозможно */
}
```

```
int *f(void)
{
    static int A[5] = { 1, 2, 3, 4, 5 };
    return A; /* ошибок нет */
}
```

К сожалению, второй способ плох по другой причине: программу, в которой функции возвращают адреса из статически выделенной памяти, сложно будет распараллелить (что будет, если чуть более сложная функция, работающая по такой же схеме, будет выполняться одновременно двумя потоками?) — потому его не рекомендуется использовать, а если функция должна возвращать массив, приходится выделять под него память при помощи `malloc` / `calloc` / `realloc`.

Про все эти вопросы мы подробно поговорим на следующем занятии.