12 Последние нововведения в C

Прогресс не остановить. Язык программирования C является стандартом ISO, постоянно пересматриваемым в попытках сделать его лучше и привнести в него новые возможности. Но это не значит, что C становится проще; по мере развития языка в нем появляются новые и сложные концепции.

В этой главе я проведу краткий обзор новшеств C11. Как вы, наверное, знаете, стандарт C11 пришел на смену C99 и позже был заменен стандартом C18. Иными словами, C18 — самая последняя версия языка C, а C11 — предыдущая.

Интересно, что в C18 не появилось никаких новых возможностей; эта версия содержит лишь исправления ошибок, найденных в C11. Таким образом, все, что мы говорим о C11, фактически относится и к C18 — то есть к самому последнему стандарту C. Как видите, в C наблюдаются постоянные улучшения... вопреки мнению о том, что этот язык давно умер!

В данной главе будет представлен краткий обзор следующих тем:

- способы определения версии C и написания кода, совместимого с разными версиями этого языка;
- новые средства оптимизации и защиты исходного кода, такие как *невозвращаемые* функции и функции *с проверкой диапазона*;
- новые типы данных и методы компоновки памяти;
- функции с обобщенными типами;
- поддержка Unicode в C11, которой не хватало в предыдущих стандартах этого языка;
- анонимные структуры и объединения;
- встроенная поддержка многопоточности и методов синхронизации в C11.

Начнем эту главу с обсуждения стандарта C11 и его нововведений.

C11

Разработка нового стандарта для технологии, которая используется на протяжении более 30 лет, — непростая задача. На C написаны миллионы (если не миллиарды!) строчек кода, и если вы хотите добавить новые возможности, то это нужно делать так, чтобы не затронуть существующий код. Новшества не должны создавать новые проблемы для имеющихся программ и не должны содержать ошибки. Такой взгляд на вещи может показаться идеалистическим, но это то, к чему нам следует стремиться.

Приведенный ниже PDF-документ находится на сайте *Open Standards* и выражает обеспокоенность и мысли участников сообщества C перед началом работы над C11: http://www.open-std.org/JTC1/SC22/wg14/www/docs/n1250.pdf. Его полезно почитать, поскольку в нем собран опыт разработки нового стандарта для языка, на котором уже было написано несколько тысяч проектов.

Мы будем рассматривать выпуск C11 с учетом всего вышесказанного. Будучи опубликованным впервые, стандарт C11 был далек от идеала и имел некоторые серьезные дефекты, со списком которых можно ознакомиться по адресу http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2244.htm.

Через семь лет после выхода C11 был представлен стандарт C18, который должен был исправить недостатки предшественника. Стоит отметить, что C18 также *неофициально* называют C17, но это один и тот же стандарт. На странице, приведенной в ссылке выше, можно просмотреть перечень дефектов и их текущее состояние. Если состояние дефекта помечено как C17, то это значит, он был исправлен в рамках C18. Это показывает, насколько сложным и щепетильным может быть процесс формирования стандарта с таким большим количеством пользователей, как у языка C.

В следующих разделах речь пойдет о новых возможностях C11. Но прежде, чем мы по ним пройдемся, необходимо убедиться в том, что у нас есть компилятор, совместимый с данным стандартом. Об этом мы позаботимся в следующем разделе.

Определение поддерживаемой версии стандарта C

На момент написания этих строк с момента выхода стандарта C11 прошло почти восемь лет. И потому было бы логично ожидать, что он уже поддерживается многими компиляторами. И это действительно так. Открытые компиляторы, такие как `gcc` и `clang`, имеют полную поддержку C11, но при необходимости могут переключаться на C99 и даже более ранние версии C. В данном разделе я покажу, как с помощью специального макроса определить версию C и в зависимости от нее использовать поддерживаемые возможности языка.

Если ваш компилятор поддерживает разные версии стандарта C, то первым делом нужно проверить, какая версия является текущей. Каждый стандарт C определяет

специальный макрос, позволяющий сделать это. До сих пор мы использовали `gcc` в Linux и `clang` в macOS. В `gcc` 4.1 C11 предоставляется в качестве одного из поддерживаемых стандартов.

Рассмотрим следующий пример, чтобы понять, как на этапе выполнения узнать текущую версию стандарта C, используя уже определенный макрос (листинг 12.1).

Листинг 12.1. Определение версии стандарта C (ExtremeC_examples_chapter12_1.c)

```
#include <stdio.h>

int main(int argc, char** argv) {
#if __STDC_VERSION__ >=  201710L
  printf("Hello World from C18!\n");
#elif __STDC_VERSION__ >= 201112L
  printf("Hello World from C11!\n");
#elif __STDC_VERSION__ >= 199901L
  printf("Hello World from C99!\n");
#else
  printf("Hello World from C89/C90!\n");
#endif
  return 0;
}
```

Как видите, данный код может различать разные версии стандарта C. Чтобы продемонстрировать, как разные версии приводят к разному выводу, скомпилируем этот исходный код несколько раз с применением разных стандартов C, поддерживаемых компилятором.

Чтобы заставить компилятор использовать определенный стандарт C, ему нужно передать параметр `-std=CXX`. Взгляните на следующую команду и на вывод, который она генерирует (терминал 12.1).

Терминал 12.1. Компиляция примера 12.1 с помощью разных версий стандарта C

```
$ gcc ExtremeC_examples_chapter12_1.c -o ex12_1.out
$ ./ex12_1.out
Hello World from C11!
$ gcc ExtremeC_examples_chapter12_1.c -o ex12_1.out -std=c11
$ ./ex12_1.out
Hello World from C11!
$ gcc ExtremeC_examples_chapter12_1.c -o ex12_1.out -std=c99
$ ./ex12_1.out
Hello World from C99!
$ gcc ExtremeC_examples_chapter12_1.c -o ex12_1.out -std=c90
$ ./ex12_1.out
Hello World from C89/C90!
$ gcc ExtremeC_examples_chapter12_1.c -o ex12_1.out -std=c89
$ ./ex12_1.out
Hello World from C89/C90!
$
```

Как видите, в новых компиляторах по умолчанию используется C11. В более старых версиях для включения C11 может понадобиться параметр `-std`. Обратите внимание на комментарии в начале файла. Я использовал многострочный формат, `/* ... */`, вместо однострочного, `//`. Дело в том, что однострочные комментарии не поддерживались в стандартах, предшествовавших C99. Поэтому пришлось сделать комментарии многострочными, чтобы код компилировался со всеми версиями C.

Удаление функции gets

Из C11 была убрана знаменитая функция `gets`. Она была подвержена атакам с *переполнением буфера*, и в предыдущих версиях ее решили сделать *нерекомендуемой*. Позже она была удалена в рамках стандарта C11. Следовательно, старый исходный код, в котором используется эта функция, нельзя скомпилировать с помощью C11.

Вместо `gets` можно использовать функцию `fgets`. Вот отрывок из справочной страницы `gets` в macOS.

Соображения безопасности

Функция gets() не подходит для безопасного использования. Ввиду отсутствия проверки диапазона и неспособности вызывающей программы надежно определить длину следующей входной строчки, применение этой функции позволяет недобросовестным пользователям вносить произвольные изменения в функциональность запущенной программы с помощью атаки переполнения буфера. В любых ситуациях настоятельно рекомендуется задействовать функцию fgets() (см. FSA).

Изменения в функции fopen

Функция `fopen` обычно используется для открытия файла и возвращения его дескриптора. Понятие *файла* в Unix очень абстрактно и может не иметь ничего общего с файловой системой. Функция `fopen` имеет следующие сигнатуры (листинг 12.2).

Листинг 12.2. Различные сигнатуры функций семейства fopen

```
FILE* fopen(const char *pathname, const char *mode);
FILE* fdopen(int fd, const char *mode);
FILE* freopen(const char *pathname, const char *mode, FILE *stream);
```

Все сигнатуры, приведенные выше, принимают входной параметр `mode`. Это строка, которая определяет режим открытия файла. В терминале 12.2 приведено описание,

взятое из справочной страницы `fopen` в FreeBSD. В нем объясняется, как следует использовать `mode`.

Терминал 12.2. Отрывок из справочной страницы fopen в FreeBSD

```
$ man 3 fopen
...
The argument mode points to a string beginning with one of the following letters:

     "r"     Open for reading. The stream is positioned at the beginning
             of the file. Fail if the file does not exist.

     "w"     Open for writing. The stream is positioned at the beginning
             of the file. Create the file if it does not exist.

     "a"     Open for writing. The stream is positioned at the end of
             the file. Subsequent writes to the file will always end up
             at the then current end of file, irrespective of
             any intervening fseek(3) or similar. Create the file
             if it does not exist.

     An optional "+" following "r", "w", or "a" opens the file
     for both reading and writing. An optional "x" following "w" or
     "w+" causes the fopen() call to fail if the file already exists.
     An optional "e" following the above causes the fopen() call to set
     the FD_CLOEXEC flag on the underlying file descriptor.
     The mode string can also include the letter "b" after either
     the "+" or the first letter.
...
$
```

Режим `x`, описанный в данном отрывке, был представлен вместе со стандартом C11. Чтобы открыть файл для записи, функции `fopen` нужно передать режим `w` или `w+`. Но проблема вот в чем: если файл уже существует, то режимы `w` и `w+` сделают его пустым.

Поэтому если программист хочет добавить что-то в файл, не стирая имеющееся содержимое, то должен задействовать другой режим, `a`. Следовательно, перед вызовом `fopen` ему нужно проверить существование файла, используя API файловой системы, такой как `stat`, и затем выбрать подходящий режим в зависимости от результата. Но теперь программист может сначала попробовать режим `wx` или `wx+`, и если файл уже существует, то `fopen` вернет ошибку. После этого можно продолжить, применяя режим `a`.

Таким образом, открытие файла требует меньше шаблонного кода, поскольку нам больше не нужно проверять его существование с помощью API файловой системы. Теперь файл можно открыть в любом режиме, используя одну лишь функцию `fopen`.

В C11 также появился API `fopen_s`. Это безопасная версия `fopen`. Согласно документации, которая находится по ссылке https://en.cppreference.com/w/c/io/fopen, функция `fopen_s` выполняет дополнительную проверку предоставленных ей буферов и их границ, что позволяет обнаружить любые несоответствия.

Функции с проверкой диапазона

Программам на языке C, которые работают с массивами строк и байтов, присуща одна серьезная проблема: они могут легко выйти за пределы диапазона, определенного для буфера или байтового массива.

Напомню, буфер — область памяти, которая служит для хранения массива байтов или строковой переменной. Выход за ее границы приводит к *переполнению буфера*, чем могут воспользоваться злоумышленники, чтобы организовать атаку (которую обычно называют *атакой переполнения буфера*). Это приводит либо к *отказу в обслуживании* (denial of service, DoS), либо к *эксплуатации* атакуемой программы.

Большинство таких атак обычно начинаются с функции, которая работает с массивами символов или байтов. В число *уязвимых* попадают функции обработки строк наподобие `strcpy` и `strcat`, находящиеся в `string.h`. У них нет механизмов проверки границ, которые могли бы предотвратить атаки переполнения буфера.

Однако в C11 появился новый набор функций *с проверкой диапазона*. Они имеют те же имена, что и функции для работы со строками, но с суффиксом `_s` в конце. Он означает, что они являются *безопасной* (secure) разновидностью традиционных функций и проводят дополнительные проверки на этапе выполнения, защищаясь от уязвимостей. Среди функций с проверкой диапазона, появившихся в C11, можно выделить `strcpy_s` и `strcat_s`.

Эти функции принимают дополнительные аргументы для входных буферов, которые исключают выполнение опасных операций. Например, функция `strcpy_s` имеет следующую сигнатуру (листинг 12.3).

Листинг 12.3. Сигнатура функции strcpy_s

```
errno_t strcpy_s(char *restrict dest, rsize_t destsz, const char *restrict src);
```

Как видите, второй аргумент — длина буфера `dest`. С его помощью функция проводит определенные проверки на этапе выполнения; например, она убеждается в том, что строка `src` не длиннее буфера `dest`, предотвращая тем самым запись в невыделенную память.