6 Динамически выделяемая память

В главе 2 вы узнали, что у каждого объекта есть срок хранения, определяющий его время жизни, и что в языке C эти сроки делятся на четыре категории: автоматические, статические, потоковые и динамические. В этой главе речь пойдет о *динамически выделяемой памяти*, которая выделяется из кучи во время выполнения. Эта память подходит в ситуациях, когда точные размеры хранимых объектов становятся известны только во время выполнения программы.

Сначала будут описаны различия между автоматическим, статическим, потоковым и динамическим сроками хранения. Мы пропустим потоковый срок хранения, поскольку это относится к параллельному выполнению, которое в данной книге не рассматривается. Затем исследуем функции, позволяющие выделять и освобождать память, а также распространенные ошибки, возникающие в процессе выделения, и то, как их можно избежать. В этой главе, как и на практике, термины *«память»* и *«хранилище»* считаются взаимозаменяемыми.

Срок хранения

Объекты могут занимать место в *хранилище* — памяти с произвольным доступом (RAM), памяти с доступом только для чтения (ROM) или регистрах. Динамический срок хранения по своим свойствам существенно отличается от автоматического и статического. Для начала рассмотрим автоматический и статический сроки хранения, которые были описаны еще в главе 2.

Объекты с автоматическим сроком хранения объявляются внутри блока или в виде параметров функции. Время жизни этих объектов начинается и заканчивается вместе с выполнением блока, в котором они объявлены. Если блок вызывается рекурсивно, то каждый раз создается новый объект с отдельным местом в памяти.

Объекты, объявленные в области видимости файла, имеют статический срок хранения. Они существуют на протяжении всей работы программы, а их хранимые значения инициализируются еще до ее запуска. Вы также можете объявить переменную в области видимости блока со статическим сроком хранения, используя спецификатор класса хранения `static`.

Куча и менеджеры памяти

Динамически выделяемая память имеет *динамический срок хранения*. Время жизни динамически выделенного объекта находится в промежутке между его выделением и освобождением. Этот вид памяти выделяется из *кучи*. Она представляет собой один или несколько крупных блоков памяти, которые могут быть поделены на части и управляются менеджером памяти.

Менеджеры памяти (memory manager) — это библиотеки для управления кучей, предоставляющие реализации стандартных функций по работе с памятью, которые будут описаны ниже. Менеджер памяти выполняется в рамках клиентского процесса. Он запрашивает у операционной системы один или несколько блоков памяти и затем выделяет их для клиентского кода, когда тот вызывает функцию выделения памяти.

Менеджеры памяти работают лишь с еще не выделенной и с уже освобожденной памятью. Как только память была выделена и до тех пор, пока она не освобождена, ее управлением занимается вызывающая сторона. Именно она отвечает за ее освобождение, хотя в большинстве реализаций это происходит автоматически при завершении программы.

Реализации менеджеров памяти

Менеджеры памяти обычно реализуют один из вариантов алгоритма выделения динамической памяти, описанного Дональдом Кнутом (1997). Этот алгоритм использует *теги границ* — поля, которые находятся по обе стороны блока памяти, возвращенного программисту, и описывают его размер. Информация

о размере позволяет перебрать все блоки памяти, начиная с любого известного блока и двигаясь в любом направлении; благодаря этому менеджер памяти может объединить два смежных свободных блока в один, чтобы минимизировать фрагментацию.

Фрагментация возникает, когда выделение и освобождение памяти приводит к появлению множества мелких и нехватке крупных блоков. В таком случае выделить большой блок памяти оказывается невозможным, хотя суммарного количества свободной памяти должно было бы хватить. Память, выделяемая клиентскому процессу и предназначенная для использования внутри менеджеров, находится в рамках единого адресного пространства клиентского процесса.

Когда следует использовать динамически выделяемую память

Динамически выделяемая память используется, когда точные размеры хранимых объектов становятся известны только во время выполнения программы. Она менее эффективна по сравнению со статически выделяемой, так как менеджеру памяти приходится искать в куче блоки подходящего размера во время выполнения, а затем, когда они больше не нужны, вызывающая сторона должна явно освобождать их. Все это требует дополнительных вычислений. Объекты, размеры которых известны на этапе компиляции, по умолчанию следует объявлять с автоматическим или статическим сроком хранения.

Когда динамически выделенную память не возвращают менеджеру памяти, происходит *утечка памяти*. Если эти утечки станут значительными по объему, то менеджер памяти рано или поздно потеряет возможность выделять новые блоки. Кроме того, динамически выделяемая память требует дополнительных вычислительных ресурсов для вспомогательных операций, таких как *дефрагментация* (консолидация смежных свободных блоков). К тому же в целях обеспечения работы этих процессов менеджер памяти зачастую использует дополнительное место, где хранит свои управляющие структуры.

Динамически выделяемая память обычно используется, когда размер или количество объектов неизвестны на этапе компиляции. Например, с помощью памяти этого типа во время выполнения программы можно прочитать таблицу из файла, особенно если вы не знаете заранее, сколько

в ней строк. Похожим образом динамически выделяемую память можно использовать для создания связных списков, хеш-таблиц, двоичных деревьев и других структур данных, состоящих из элементов, количество которых неизвестно в момент компиляции.

Функции для управления памятью

Стандартная библиотека C предоставляет функции для выделения и освобождения динамической памяти. В их число входят `malloc`, `aligned_alloc`, `calloc` и `realloc`. Освобождение можно выполнить с помощью функции `free`. В OpenBSD есть функция `reallocarray`, которая не является частью стандартной библиотеки, но тоже может пригодиться для выделения памяти.

Функция malloc

Функция `malloc` выделяет место для объекта заданного размера, начальное значение которого невозможно определить. В листинге 6.1 мы вызываем функцию `malloc`, чтобы динамически выделить место для объекта размера `struct widget`.

Листинг 6.1. Выделение места для widget с помощью функции malloc

```
#include <stdlib.h>
typedef struct {
  char c[10];
  int i;
  double d;
} widget;

❶ widget *p = malloc(sizeof(widget));
❷ if (p == NULL) {
    // Обрабатываем ошибку выделения
  }
  // Продолжаем обработку
```

Все функции для выделения памяти принимают аргумент типа `size_t`, определяющий количество байтов, которые нужно выделить ❶. В целях переносимости при вычислении размера объектов мы используем операцию `sizeof`, поскольку в разных реализациях различные типы, такие как `int` и `long`, могут иметь разную разрядность.

Функция `malloc` возвращает либо нулевой указатель, чтобы сообщить об ошибке, либо указатель на выделенный участок. В связи с этим мы проверяем, возвращает ли `malloc` нулевой указатель ❷, и обрабатываем ошибку соответствующим образом.

После того как функция успешно вернет выделенный участок, мы можем обращаться к членам структуры `widget`, используя указатель `p`. Например, `p->i` позволяет обратиться к члену `widget` типа `int`, а `p->d` предоставляет доступ к члену типа `double`.

Выделение памяти без объявления типа

Значение, которое возвращает `malloc`, можно хранить в виде указателя на `void`, чтобы не объявлять тип объекта, на который тот ссылается:

```
void *p = malloc(size);
```

Кроме того, вы можете использовать указатель на `char`, как это было принято делать до появления в C типа `void`:

```
char *p = malloc(size);
```

В обоих случаях у объекта, на который указывает `p`, нет типа, пока он не скопирован в выделенный участок. Когда это происходит, объекту назначается *фактический тип* последнего объекта, скопированного в данный блок памяти. В следующем примере блок, на который указывает `p`, получает фактический тип `widget` после вызова `memcpy`.

```
widget w = {"abc", 9, 3.2};
memcpy(p, &w, sizeof(widget));     // приведено к указателям void *
printf("p.i = %d.\n", p->i);
```

Поскольку в выделенном блоке можно хранить объекты любых типов, мы можем направлять указатели, возвращаемые любыми функциями выделения памяти, включая `malloc`, на любой тип объектов. Например, если реализация поддерживает объекты с 1-, 2-, 4-, 8- и 16-байтными выравниваниями, то при выделении 16 или больше байт памяти возвращаемый указатель будет иметь выравнивание, кратное 16.

Приведение указателя к типу объявленного объекта

Даже опытные программисты на C имеют разные мнения о том, нужно ли приводить указатель, возвращенный функцией `malloc`, к типу объявлен-

ного объекта. Следующий оператор присваивания приводит указатель к типу `widget *`:

```
widget *p = (widget *)malloc(sizeof(widget));
```

Строго говоря, это приведение типов не является обязательным. Язык C позволяет косвенно преобразовать указатель типа `void` (который вернула функция `malloc`) в указатель на объект любого типа с подходящим выравниванием (в противном случае поведение неопределено). Ручное приведение результата `malloc` к нужному типу дает возможность компилятору обнаружить непреднамеренные преобразования указателей, а также несоответствие размеров выделяемого блока и типа объекта указателя в выражении приведения.

В примерах, приводимых в данной книге, обычно используется ручное приведение типов, но оба стиля являются приемлемыми. Более подробную информацию об этом можно найти в правиле MEM02-C стандарта CERT C (немедленно приводите результат вызова функции для выделения памяти к указателю на выделенный тип).

Чтение неинициализированной памяти

Содержимое памяти, возвращенной из `malloc`, *не инициализировано*. Это значит, в нем находятся неопределенные значения. Чтение неинициализированной памяти всегда плохая идея, и такую операцию следует считать неопределенным поведением. Если хотите узнать больше, то рекомендую обратиться к моей подробной статье о *неинициализированном чтении* (Сикорд, 2017). Функция `malloc` не инициализирует возвращаемую память, поскольку ожидается, что вы все равно ее перезапишете.

Тем не менее многие новички ошибочно предполагают, что память, возвращенная из `malloc`, содержит нули. Программа, представленная в листинге 6.2, допускает именно эту ошибку.

Листинг 6.2. Ошибка инициализации

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>


int main(void) {
  char *str = (char *)malloc(16);
```

```
  if (str) {
    strncpy(str, "123456789abcdef", 15);
    printf("str = %s.\n", str);
    free(str);
    return EXIT_SUCCESS;
  }
  return EXIT_FAILURE;
}
```

Эта программа выделяет 16 байт памяти, вызывая `malloc`, и затем использует `strncpy`, чтобы скопировать первые 15 байт строки в выделенный блок. Программист пытается создать корректную строку с нуль-символом в конце, копируя на один байт меньше, чем размер выделенной памяти. Делая это, программист предполагает, что в выделенном блоке уже содержится значение 0, которое будет служить нулевым байтом. Но вместо этого там вполне может находиться ненулевое значение, и в этом случае строка не будет оформлена как следует, а вызов `printf` приведет к неопределенному поведению.

Распространенное решение этой проблемы состоит в том, чтобы записать нуль-символ в последний байт выделенного блока, как показано ниже:

```
strncpy(str, "123456789abcdef", 15);
str[15] = '\0';
```

Если исходная строка меньше 15 байт, то будет скопирован нуль-символ и присваивание в строчке ❶ окажется лишним. Если же исходная строка занимает 15 или больше байт, то наличие этого присваивания будет гарантировать, что она заканчивается нуль-символом.

Функция aligned_alloc

Функция `aligned_alloc` похожа на `malloc`, но вместе с размером выделяемого объекта требует также указать его выравнивание. Ниже представлена ее сигнатура, где `size` определяет размер объекта, а `alignment` — выравнивание:

```
void *aligned_alloc(size_t alignment, size_t size);
```

Появление функции `aligned_alloc` в стандарте C11 объясняется тем, что некоторое аппаратное обеспечение имеет более строгие требования к выравниванию памяти. И хотя язык C требует, чтобы память, динамически выделяемая из `malloc`, была достаточно выровнена для всех стандартных типов, включая массивы и структуры, иногда может возникнуть необхо-

димость в переопределении тех решений, которые компилятор принимает по умолчанию.

Функция `aligned_alloc`, как правило, используется с целью задать более строгое выравнивание по сравнению со стандартным (то есть с большей степенью двойки). Если значение `alignment` не является корректным выравниванием, которое поддерживается реализацией, то данная функция завершается неудачно и возвращает нулевой указатель. Больше о выравнивании можно узнать в главе 2.

Функция calloc

Функция `calloc` выделяет место для массива с `nmemb` объектами, каждый из которых занимает `size` байт. Ее сигнатура выглядит так:

```
void *calloc(size_t nmemb, size_t size);
```

Данная функция заполняет блок памяти нулевыми байтами. Они могут отличаться от значения, которое используется для представления нуля в типах с плавающей запятой или константных нулевых указателей. Функция `calloc` также позволяет выделить место и для одиночного объекта, который в этом случае можно считать массивом, состоящим из одного элемента.

Внутри функция `calloc` умножает `nmemb` на `size`, чтобы определить количество байтов, которое нужно выделить. По историческим причинам некоторые реализации `calloc` не проверяли, переполняются ли эти значения при умножении. Однако современные версии `calloc` выполняют данную проверку, и если произведение нельзя представить с помощью типа `size_t`, то возвращают нулевой указатель.

Функция realloc

Функция `realloc` увеличивает или уменьшает размер ранее выделенного блока памяти. Она принимает указатель на некий блок, выделенный одним из предыдущих вызовов `aligned_alloc`, `malloc`, `calloc` или `realloc` (или нулевой указатель) и размер. Ее сигнатура выглядит так:

```
void *realloc(void *ptr, size_t size);
```

Вы можете использовать функцию `realloc` для расширения или (что случается реже) сужения массива.

Борьба с утечками памяти

Чтобы не наделать ошибок при использовании `realloc`, вы должны понимать (на концептуальном уровне), как реализована эта функция. Обычно она вызывает `malloc` для выделения нового участка памяти, а затем копирует в него содержимое старого участка так, чтобы не превысить ни старый, ни новый размер. Если новый участок больше старого, то функция `realloc` оставляет лишнее место неинициализированным. Если ей удается выделить новый объект, то она вызывает `free`, чтобы освободить старый. В случае неудачи `realloc` сохраняет данные старого объекта по тому же адресу и возвращает нулевой указатель. Причиной неудачного вызова `realloc` может быть, к примеру, нехватка памяти, доступной для выделения запрошенного количества байтов. Следующий пример использования `realloc` содержит ошибку:

```
size += 50;
if ((p = realloc(p, size)) == NULL) return NULL;
```

В этом примере переменная `size` инкрементируется на 50, после чего вызывается `realloc`, чтобы увеличить размер блока, на который указывает `p`. Если вызов `realloc` завершается неудачно, то указателю `p` присваивается значение `NULL`, однако блок, на который указывает `p`, не освобождается, что приводит к утечке памяти.

В листинге 6.3 показано, как правильно использовать функцию `realloc`.

Листинг 6.3. Правильное использование функции realloc

```
void *p2;
void *p = malloc(100);
//---snip---
if ((nsize == 0) || (p2 = realloc(p, nsize)) == NULL) {
  free(p);
  return NULL;
}
p = p2;
```

В данном фрагменте кода объявляются две переменные, `p` и `p2`. Первая указывает на динамически выделенную память, которую вернула функция `malloc`, а вторая изначально не инициализируется. В конечном счете мы меняем размер этой памяти, вызывая функцию `realloc` с указателем на `p` и новым размером `nsize`. Значение, возвращаемое из `realloc`, присваивается переменной `p2`, чтобы не перезаписывать указатель, хранящийся в `p`. Если `realloc` возвращает нулевой указатель, то память, на которую указывает `p`, освобождается и функция возвращает `NULL`. Если все про-

шло хорошо, и realloc возвращает указатель на блок размером nsize, то переменной p присваивается указатель на вновь выделенный блок и выполнение продолжается.

Данный код также включает проверку на выделение нуля байт. Функции realloc не следует передавать 0 в качестве аргумента size, поскольку это фактически (а в C2x официально) является неопределенным поведением.

Следующий вызов функции realloc не возвращает нулевой указатель, однако адрес, который хранится в p, становится недействительным и читать его недопустимо:

```
newp = realloc(p, ...);
```

В частности, следующая проверка является недопустимой:

```
if (newp != p) {
  // обновляем указатели с учетом заново выделенной памяти
}
```

После вызова realloc, вне зависимости от того, менял этот вызов адрес выделенного блока или нет, любые указатели на память, на которую ранее ссылался указатель на p, должны быть перенаправлены к блоку, на который ссылается newp.

Чтобы решить эту проблему, можно ввести дополнительную абстракцию, иногда называемую *дескриптором (handle)*. Если указатель всегда используется косвенно, то при его перенаправлении будут автоматически обновлены все участки кода, в которых он фигурирует.

Вызов realloc с нулевым указателем

Вызов realloc с нулевым указателем эквивалентен вызову malloc. При условии, что значение newsize не равно 0, следующий код:

```
if (p == NULL)
  newp = malloc(newsize);
else
  newp = realloc(p, newsize);
```

равнозначен:

```
newp = realloc(p, newsize);
```

Первая, более длинная версия кода вызывает malloc при первом выделении памяти и realloc, если позже потребуется изменить размер. Но поскольку

вызов `realloc` с нулевым указателем эквивалентен вызову `malloc`, вторая версия делает то же самое, только в сжатом виде.

Функция reallocarray

Функция `reallocarray` в OpenBSD может заново выделить память для массива и при этом проверяет на переполнение, когда вычисляет его размер. Таким образом, вам не нужно проводить данную проверку самостоятельно. Сигнатура функции `reallocarray` выглядит так:

```
void *reallocarray(void *ptr, size_t nmemb, size_t size);
```

Функция `reallocarray` выделяет память для `nmemb` размера `size` и проверяет, не происходит ли целочисленное переполнение при вычислении `nmemb * size`. Другие платформы, включая GNU C Library (libc), тоже внедрили у себя эту функцию, и теперь ее предлагают включить в следующую версию стандарта POSIX. Стоит отметить, что `reallocarray` не обнуляет выделенную память.

Как мы уже видели в предыдущих главах, целочисленное переполнение — это серьезная проблема, которая может приводить к переполнениям буферов и другим уязвимостям безопасности. Например, в следующем коде выражение `num * size` может переполниться до того, как будет передано функции `realloc` в качестве аргумента `size`:

```
if ((newp = realloc(p, num * size)) == NULL) {
  //---snip---
```

Функция `reallocarray` подходит в ситуациях, когда для определения размера выделяемого блока умножаются два значения:

```
if ((newp = reallocarray(p, num, size)) == NULL) {
  //---snip---
```

Если `num * size` грозит переполнением, то этот вызов `reallocarray` завершится неудачно и вернет нулевой указатель.

Функция free

Когда динамически выделенная память больше не нужна, ее следует освободить путем вызова функции `free`. Освобождение памяти играет важную роль, позволяя повторно задействовать одни и те же блоки. Это снижает