Пояснительная бригада по распределению памяти

В языке программирования С можно реализовать три основных способа выделения памяти:

— Статическое выделение памяти;
— Автоматическое выделение памяти;
— Динамическое выделение памяти.

Структура программы на языке Си приведена на рис. 1.

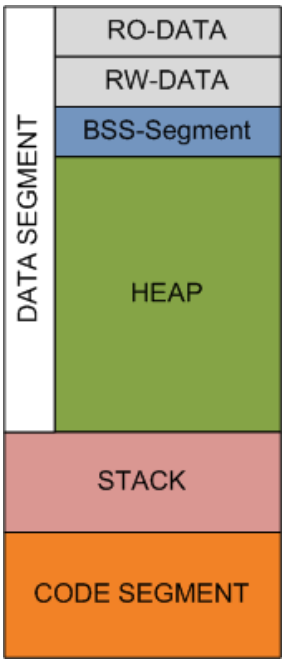

Рис. 1. Структура памяти программы на Си

1. Статическое выделение памяти — это когда память для какой-либо программы распределяется единожды при ее запуске и это выделенное количество памяти не меняется, пока программа не закончит работу.

Время существования: всё время выполнения программы.

Инициализация: происходит один раз до запуска программы.

Размер: фиксированный, неизменяемый.

Типичное размещение: отдельный сегмент памяти (DATA). DATA состоит из статических и глобальных переменных, которые явно инициализируются значениями. Этот сегмент может быть разбит:

— ro-data (read only data) – сегмент данных только для чтения,
— rw-data (read write data) – сегмент данных для чтения и записи.

BSS-сегмент (block started by symbol) содержит неинициализированные глобальные переменные, или статические переменные без явной инициализации.

> Обычно загрузчик программ инициализирует BSS область при загрузке приложения нулями. Дело в том, что в DATA области переменные инициализированы – то есть затирают своими значениями выделенную область памяти. Так как переменные в BSS области не инициализированы явно, то они теоретически могли бы иметь значение, которое ранее хранилось в этой области, **а это уязвимость** которая предоставляет доступ до, возможно, приватных данных (см. информационная безопасность). Поэтому загрузчик вынужден обнулять все значения. За счёт этого и неинициализированные глобальные переменные, и статические переменные по умолчанию равны нулю.

Размер доступной статической памяти зависит от ОС, и он ограничен. К примеру, для Windows, он ограничен 2-мя гигабайтами. В целом, такого количества памяти достаточно для большинства программ.

Пример 1.1.

```
// глобальные переменные, их видно во всех функциях программы
int var;
float mass[100];
static char name[30];

int main() {
  static float eps = 0.001;
  const char* string = "hello world";

  /* some actions */

  return 0;
}
```

Все переменные в данном случае являются статически выделенными. Переменные var, mass, name более того являются глобальными переменными. Переменная string хранится в ro-data, в отличие от остальных переменных.

2. Автоматическое выделение памяти — это когда память распределяется для отдельных параметров программы в момент их «входа» в работу, после окончания работы этих отдельных параметров память высвобождается; такая память выделяется столько раз, сколько необходимо.

Выполняется, например, для параметров функции и локальных переменных. Память выделяется при входе в блок, в котором находятся эти переменные, и удаляется при выходе из него.

Время существования: блок, в котором объявлен объект.

Инициализация: отсутствует в случае отсутствия явной инициализации.

Размер: фиксированный, неизменяемый.

Типичное размещение: стек / регистры процессора. **Стек вызовов** обычно растёт "навстречу" куче, то есть с увеличением стека адрес вершины стека уменьшается (съедая память из кучи). Набор значений, которые кладутся на стек одной функцией, называются **фреймом**. Каждый раз, когда мы создаём локальные переменные, они располагаются на стеке. После выхода из функции стек освобождается от фрейма. Как только мы выходим из функции, стек очищается и переменные исчезают.

> Один фрейм хранит как минимум одно значение - адрес возврата. Вызов функций и передача параметров также происходит через стек.

Пример 2.1.

```
int add(int x, int y) {
  return x + y;
}

int main() {
  static int var1, var2;

  static int sum = add(var1, var2);

  return 0;
}
```

Переменные x и y – автоматически выделенные переменные, они существуют только внутри функции add, после её завершения переменные освобождают, выделенную под них память. Остальные переменные статические.

Пример 2.2.

```
#include <stdio.h>

#define SIZE 10

int main() {
  int mass[SIZE];
  int i = 0;

  for (i = 0; i < SIZE; i++)
    mass[i] = i + 2 * i;

  for (int ii = 0; ii < SIZE; ii++)
    printf("%d ", mass[ii]);

  return 0;
}
```

Все переменные в данном коде являются автоматически выделенными, однако заметим, что переменная ii существует только внутри указанного цикла. Остальные существую от их определения и до конца работы функции main().

Пример 2.3.

```
#include <stdio.h>

#define SIZE 10

int main() {
  float mass[SIZE];
  int i = 0;

  for (int i = 0; i < SIZE; i++) {
    float shift = 0.01;
    mass[i] = i + 2 * i + shift * (i + 3);
  }

  for (int ii = 0, shift = 2; ii < SIZE; ii+= shift) {
    printf("%f ", mass[ii]);
  }

  static char shift = '\n';
  printf("%c", shift);

  /* some actions */

  return 0;
}
```

Переменная shift в данном коде объявлена трижды. В первом случае это автоматически выделенная переменная типа float, во втором случае автоматически выделенная переменная типа int, а в третьем случае это статическая переменная для хранения символа перехода на новую строку. После данного объявления имя переменной уже не может быть использовано для последующей локальной переменной, она будет существовать до конца работы программы.

3. Динамическое выделение памяти — это когда заранее не известен объем памяти, который может понадобиться для выполнения программы, поэтому мы «просим» операционную систему зарезервировать какую-то часть памяти для нашего программного продукта, чтобы потом использовать ее нужное количество.

Функции для динамического распределения памяти (из стандартной библиотеки языка <stdlib.h> или <malloc.h>):

— malloc (от англ. memory allocation, выделение памяти),
— calloc (от англ. clear allocation, чистое выделение памяти),
— realloc (от англ. reallocation, перераспределение памяти),
— free (англ. free, освободить).

Прототипы функций:

```
void *malloc(size_t size);
void *calloc(size_t count, size_t elem_size);
void *realloc(void *memory, size_t newSize);
void free(void *memory).
```

Для точного определения размера массива в байтах рекомендуется использование функции `int sizeof(тип);`

Время существования: от вызова malloc до вызова free.

Инициализация: частично в случае calloc.

Размер: любой, изменяемый.

Типичное размещение: куча. Куча – начинается за BSS сегментом и начиная оттуда растёт, соответственно с увеличением адреса. Этот участок используется для выделения на нём памяти с использованием функции malloc (и прочих) и для очисти с помощью функции free.

Функция malloc выделяет память на куче по определённому адресу, после чего возвращает его. Теперь указатель p хранит этот адрес и может им пользоваться для работы.

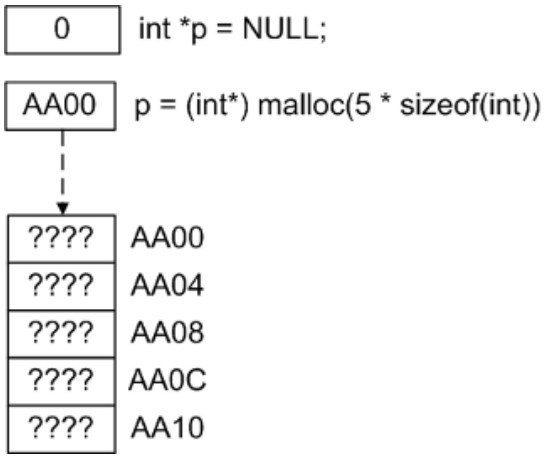

Рис. 2. Выделение памяти под массив на 5 элементов по указателю p

Пример 3.1.

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

int main() {
  static const int size = 100;
  int *mass = NULL;

  mass = (int*)malloc(size * sizeof(int));

  for (int i = 0; i < size; i++)
    mass[i] = i * i;

  for (int i = 0; i < size; i++)
    printf("%d ", mass[i]);

  free(mass);
  return 0;
}
```

Функция free освобождает память, но при этом не изменяет значение указателя (см. рис. 3), о чём нужно помнить.

Пример 3.2.

Попробуем написать программу, иллюстрирующую рис. 2.

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

int main() {
  static const int size1 = 10, size2 = 20;
  int *mass = NULL;

  mass = (int*)malloc(size1 * sizeof(int));
  printf("%p:\n", mass);

  for (int i = 0; i < size1; i++)
    mass[i] = i * i;

  for (int i = 0; i < size1; i++)
    printf("%p - %d\n", &mass[i], mass[i]);

  free(mass);

  mass = (int*)malloc(size2 * sizeof(int));
  printf("\n%p:\n", mass);

  for (int i = 0; i < size2; i++)
    mass[i] = i * i + 1;

  for (int i = 0; i < size2; i++)
    printf("%p - %d\n", &mass[i], mass[i]);

  free(mass);

  return 0;
}
```

Вывод по результату работы этой функции представлен на рис. 3.

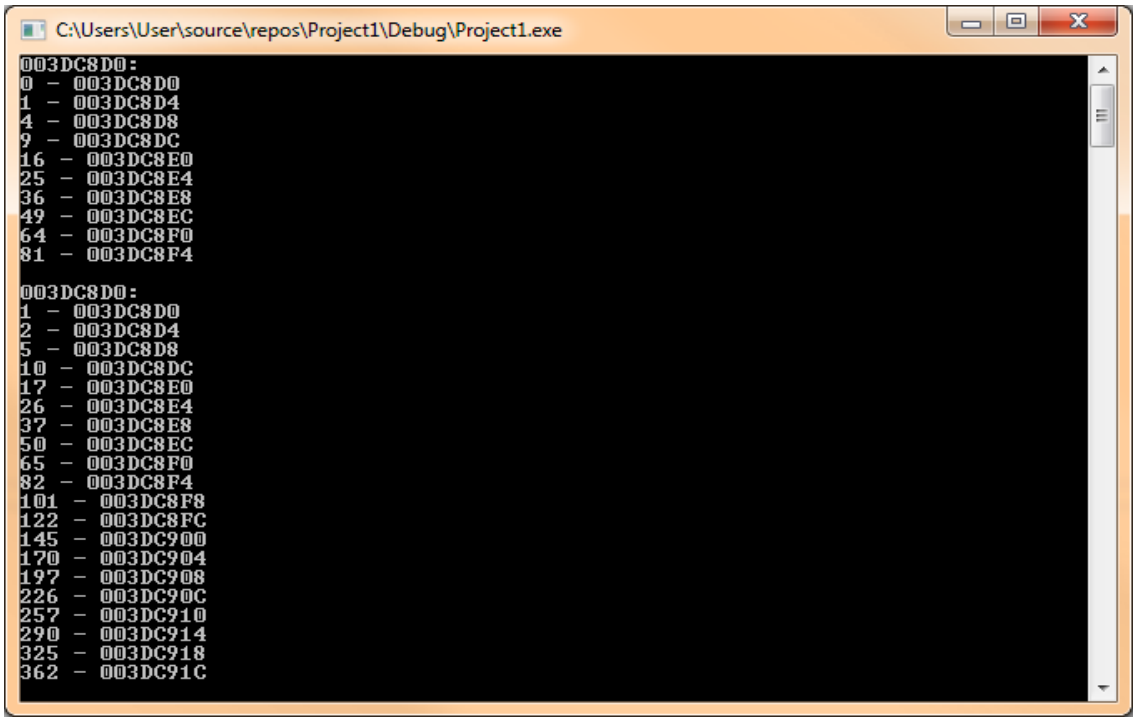

Рис. 3. Вывод массива с указанием адресов памяти

Можно создавать и динамические переменные.

Пример 3.3.

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

int main() {
  int *a = (int*)malloc(sizeof(int));
  float *b = (float*)malloc(sizeof(float));
  char *c = (char*)malloc(sizeof(char));

  printf("%p: %d\n", a, *a);
  printf("%p: %.3f\n", b, *b);
  printf("%p: %c\n", c, *c);

  (*a) = 15;
  (*b) = 0.015;
  (*c) = 'f';

  printf("%p: %d\n", a, *a);
  printf("%p: %.3f\n", b, *b);
  printf("%p: %c\n", c, *c);

  (*a) = 1;
  (*b) = 51.09;
  (*c) = 'z';

  printf("%p: %d\n", a, *a);
  printf("%p: %.3f\n", b, *b);
  printf("%p: %c\n", c, *c);

  free(a); free(b); free(c);

  return 0;
}
```

Вывод по результату работы этой функции представлен на рис. 4.

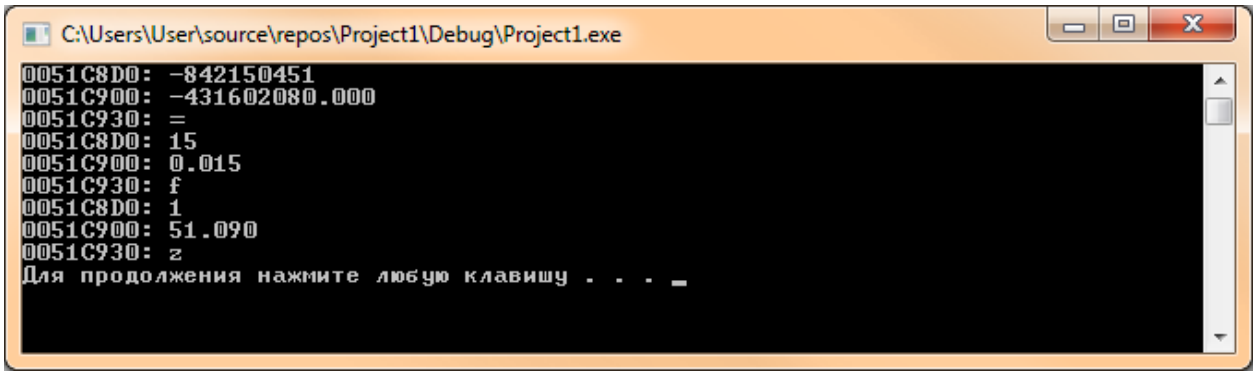

Рис. 4. Использование динамических переменных

Указатель — переменная, содержащая адрес объекта. Указатель не несет информации о содержимом объекта, а содержит сведения о том, где размещен объект.

Для работы с указателями в Си определены две операции:

- операция * (звездочка) — позволяет получить значение объекта по его адресу — определяет значение переменной, которое содержится по адресу, содержащемуся в указателе;
- операция & (амперсанд) — позволяет определить адрес переменной.

К элементам динамического массива можно обращаться не только с помощью индекса, но и по указателю.

Пример 3.4. Следующие два кода работают одинаково:

Работа с массивами по указателю	Работа с массивами по индексу

`#include <stdlib.h>`
`#include <stdio.h>`
 	
`int main() {`
`  static const int size = 10;`
`  int *mass = NULL;`
 	
`  mass = (int*)malloc(size * sizeof(int));`
`  printf("%p:\n", mass);`
 	
`  for (int i = 0; i < size; i++)`
`  *(mass + i) = i * i;`
 	
`  for (int i = 0; i < size; i++)`
`  printf("%p - %d\n", &*(mass + i),`
`*(mass + i));`
 	
`  free(mass);`
 	
`  return 0;`
`}`
	`#include <stdlib.h>`
	`#include <stdio.h>`
	
	`int main() {`
	`  static const int size = 10;`
	`  int *mass = NULL;`
	
	`  mass = (int*)malloc(size * sizeof(int));`
	`  printf("%p:\n", mass);`
	
	`  for (int i = 0; i < size; i++)`
	`  mass[i] = i * i;`
	
	`  for (int i = 0; i < size; i++)`
	`  printf("%p - %d\n", &mass[i],`
	`mass[i]);`
	
	`  free(mass);`
	
	`  return 0;`
	`}`

> При этом индекс i любого элемента двумерного массива можно получить по формуле index = i*m+j, где i - номер текущей строки; j - номер текущего столбца.
>
> То есть обращение к элементу матрицы можно выполнить с помощью *(mass+i*m+j).

Пример 3.5.

Код из примера 3.2 может быть переписан с использованием функции realloc().

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

int main() {
  static const int size1 = 10, size2 = 20;
  int *mass = NULL;

  mass = (int*)malloc(size1 * sizeof(int));
  printf("%p:\n", mass);

  for (int i = 0; i < size1; i++)
    mass[i] = i * i;

  for (int i = 0; i < size1; i++)
    printf("%p - %d\n", &mass[i], mass[i]);

  mass = (int*)realloc(mass, size2 * sizeof(int));
  printf("\n%p:\n", mass);

  for (int i = 0; i < size2; i++)
    mass[i] = i * i + 1;

  for (int i = 0; i < size2; i++)
    printf("%p - %d\n", &mass[i], mass[i]);

  free(mass);
  return 0;
}
```

При использовании realloc желательно (в некоторых средах, например, в Visual Studio, обязательно) инициализировать указатель хотя бы значением NULL. В целом можно было вместо первого malloc использовать realloc.

Размер блока памяти, на который ссылается указатель изменяется на указанное число байтов. Блок памяти может уменьшаться или увеличиваться в размере. Содержимое блока памяти сохраняется даже если новый блок имеет меньший размер, чем старый. Но отбрасываются те данные, которые выходят за рамки нового блока. Если новый блок памяти больше старого, то содержимое вновь выделенной памяти будет неопределенным.

Доп. информация о динамическом выделении памяти и работе с указателями тут.

> Переменная указатель p хранит адрес области памяти, начиная с которого она может им пользоваться. **Однако, она не хранит размера этой области.** Откуда тогда функция free знает, сколько памяти необходимо освободить?
>
> Очевидно, что информация о размере выделенного участка должна где-то храниться. Есть несколько способов решения этой проблемы:
>
> 1. Существует некоторая карта, в которой хранится размер выделенного участка. Каждый раз при освобождении памяти компьютер обращается к этим данным и получает нужную информацию.
>
> 2. Второе решение более распространено. Информация о размере хранится на куче до самих данных. Таким образом, при выделении памяти резервируется места больше и туда записывается информация о выделенном участке. При освобождении памяти функция free "подсматривает", сколько памяти необходимо удалить.

Наконец, функция free() принимает указатель на область, подлежащую освобождению. Стоит знать, что она **не проверяет указатель на правильность**, и может «освободить» невыделенную область памяти, что в некоторых реализациях может привести к необратимому повреждению кучи! Вызов функции с NULL (пустой памятью) **безопасен** (проверка на NULL обязана выполняться внутри free() согласно стандарту). Для избежания повреждения кучи некоторые руководства по языку Си рекомендуют **обнулять каждый освобождаемый указатель.**

Область памяти, освобождённая после вызова free() может быть выделена снова, однако частый вызов malloc и free() может привести к фрагментации кучи и невозможности выделить области памяти большого объёма.

> Возвращаясь к рис. 1, обратим внимание на оставшийся блок памяти. **Сегмент кода**, или текстовый сегмент, или просто текст, содержит исполняемые инструкции. У него фиксированный размер и обычно он используется только для чтения, если же в него можно писать, то архитектура поддерживает самомодификацию. Сегмент кода располагается после начала стека, поэтому в случае роста он [стек] не перекрывает сегмент кода.