8.1. Распределение памяти

Распределение памяти (*управление памятью)* - это определение места (адресов) для хранения различных объектов программы. Распределение памяти бывает статическим и динамическим.

В программировании термин "*статический*" означает "происходящий до начала исполнения программы (не меняющийся во время ее работы)", "*динамический*" - "происходящий (изменяющийся) во время работы программы".

Статическое распределение памяти производится во время трансляции программы (до начала работы) и не меняется во время ее работы.

При *динамическом распределении памяти* адреса объектов определяются и могут изменяться во время работы программы. Это замедляет работу программы, но позволяет экономить память, размещая разные объекты по очереди в одной и той же области памяти, а также использовать *динамические объекты*, структура и размеры которых изменяются динамически (т. е. во время работы программы).

Каждый объект программы (константа, переменная, массив, структура, функция, тип, метка) относится к определенному классу памяти. *Класс памяти* определяет место хранения и период существования объекта. В языке С четыре класса памяти: **auto** - *автоматический*, **static** - *статический*, **extern** - *внешний* и **register** - *регистровый*. Класс памяти может указываться перед типом объекта при его описании (определении).

определение-данных ::= [класс-памяти] тип список-описат-иниц ;

класс-памяти ::= **auto** | **static** | **extern** | **register**

По умолчанию (если класс памяти не указан) локальные объекты, (описанные внутри функций), относятся к классу **auto**, глобальные (описанные вне функций) - к классу **extern**.

Для объектов классов **static** и **extern** распределение памяти присходит статически. Разница между ними заключается в том, что объекты класса **static** доступны только тому блоку или модулю программы, в котором они описаны, а внешние объекты доступны всем модулям программы.

В языке С используются два вида динамического распределения памяти: автоматическое и управляемое. Во время работы программы (т. е. динамически) при входе в блок *автоматически* выделяется память для его объектов класса **auto**, а при выходе из блока эта память освобождается. Для автоматического распределения памяти используется стек.

Стеком называют структуру данных, в которой происходит поступление и удаление элементов по принципу “последним пришел, первым ушел”. Стек можно представить себе в виде последовательности, в которой

элементы добавляются и удаляются с одной и той же стороны, называемой *вершиной* стека. Всем знакомый пример – заднее сиденье легкового автомобиля, когда посадка и высадка разрешены только со стороны тротуара. Эта сторона является вершиной стека.

При *управляемом* распределении выделение и освобождение областей памяти производится в произвольные моменты времени работы программы по указанию программиста с помощью стандартных функций управления памятью malloc и free (и их разновидностей).

Функция malloc выделяет участок памяти, размер которого в байтах задается ей в виде параметра. Содержимое этого участка не определено. Функция malloc возвращает в качестве значения указатель на начало выделенного участка или пустой указатель NULL, если свободной памяти заданного размера не оказалось.

Функция free освобождает участок памяти (выделенный ранее функцией malloc), указатель которого является ее параметром.

Пример. Управление памятью С-программы. Ниже приведен пример С–программы. Цифры в скобках отмечают различные моменты времени.

```
      int x,y;
(1)   void main (void)
      {   int a, t[];
(2)        . . .
          {   int b;  static int z;
(3)             . . .
              t=malloc(3*sizeof(int));          /*Выделение памяти  для t */
(4)             . . .
              f(t, b);
(6)             . . .

          }
(7)       . . .
          free(t);                              /* Освобождение памяти для t */
(8)       . . .
          {  int c;
(9)           . . .

          }
 (10)     . . .
      }
(11)
      int f (int x[], int y)
      {  int d;
(5)       . . .
```

```
    return d;
}
```

На рис 8.1. показано распределение области оперативной памяти, выделенной этой программе (на примере персональной ЭВМ типа IBM PC) в различные моменты времени.

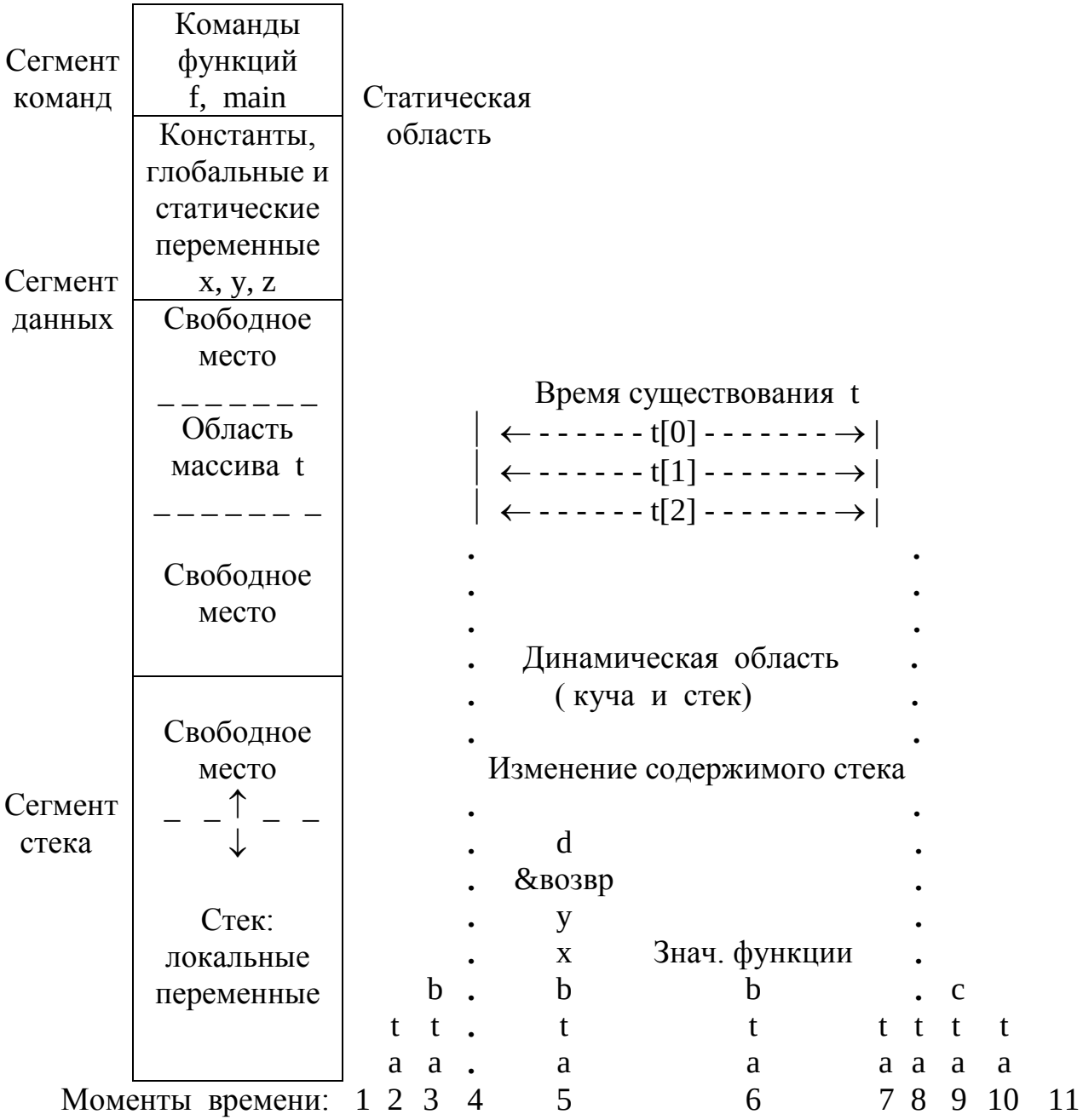

Рис. 8.1. Распределение области памяти программы

По способу управления оперативная память разделяется на *статическую область*, в которой размещаются команды и константы

программы, а также ее глобальные и статические переменные, и *динамическую область*, состоящую из стека и кучи.

В стеке память автоматически выделяется при входе в блок и освобождается при выходе из него. В *куче* (heap) по запросам программы функция malloc выделяет участки памяти, а затем функция free их освобождает – учитывает как освободившиеся.

По виду хранимой информации область оперативной памяти программы разделяется на *сегмент кода* (команд), *сегмент данных* и *сегмент стека*.

В момент (1) перед выполнением функции main в статическую область загружены команды и константы функций f и main. Здесь также выделено место для значений глобальных переменных x, y и статической переменной z.

В момент (2) после входа в функцию main в стеке выделено место для ее локальных переменных: x и адреса массива t.

В момент (3) после входа в блок в стеке выделено место для локальной переменной b этого блока. Статическая переменная z доступна только данному блоку программы. После выхода из блока в момент (7) значение z сохранится до конца работы программы, но снова стать доступным и изменяться может только при повторном входе в этот же блок. Это могло бы произойти, если бы данный блок находился внутри не показанного в программе цикла (на месте многоточия …).

После выполнения функции malloc в момент (4) в куче выделен участок для трех элементов массива, а переменной t присвоен адрес этого участка. Массив t существует и может использоваться программой до момента (8), когда функция free освободит занимаемый массивом участок памяти.

В момент (5) после входа в функцию f в стек помещены сначала значения ее параметров: x, равного адресу массива t; y, равного значению переменной b, и адреса возврата (являющегося неявным параметром). Затем выделено место для значения локальной переменной d функции f. Таким образом, во время выполнения функции f под массивом x фактически понимается массив t.

В момент (6), после возврата в основную программу из функции f, из стека удалена ее локальная переменная d, а также параметры y, x и адрес возврата. На их место помещено значение функции f, равное переменной d.

После выхода из блока в момент (7) из стека удалена локальная переменная b этого блока.

В момент (8), когда функция free освободила участок кучи, занимаемый массивом t, этот массив перестал существовать. Массив t больше использовать нельзя, хотя его адрес и сохранился в переменной t.

Локальная переменная c размещается в стеке после входа в блок в момент (9) и удаляется из стека после выхода из этого блока в момент (10).

В момент (11), когда закончится выполнение функции main, освободится вся занимаемая программой область оперативной памяти и произойдет возврат управления в операционную систему.

8.2. Рекурсия

Рекурсия - это самовложение (лат. recursio - возвращение), при котором внутри некоторого объекта содержится подобный ему объект. Примером могут служить куклы-матрешки. Подобное явление можно увидеть на экране телевизора, если в кадр попадет монитор с тем же изображением, которое идет в эфир. Этот эффект часто используют в рекламе.

В математике используются рекурсивные определения, в программировании - рекурсивные программы.

Рекурсивным называется *определение* некоторого понятия через само определяемое понятие.

Пример 1. Дано следующее определение факториала:

k! = 1 при k = 0, k! = 1 * 2 * ... * k при целом k > 0.

Это определение использует *итерацию* (повторение) умножений и его можно назвать итеративным.

Можно дать другое определение факториала:

k! = 1 при k = 0,
k! = (k - 1)! * k при целом k > 0.

Здесь факториал от k выражается через само определяемое понятие - факториал от (k - 1), т. е. второе определение является рекурсивным.

Вычислим, например, 2! по рекурсивному определению:

2! = 2 * (2-1)! = 2*1! = 2 * 1 * (1-1)! = 2*1*0! = 2*1*1 = 2

Первое определение использовано для вычисления факториала в программах 1 и 2 с помощью циклов (занятие 7).

Вычисление факториала на основе рекурсивного определения приведено в программе 8.1.

Программа 8.1

```
/*      Рекурсивное определение функции  fakt(k) = k!   */
int fakt (int k)
{  if (k > 0)  return fakt(k-1) * k;
   else  return 1;
}
```

Программа (точнее, подпрограмма) называется *рекурсивной*, если она вызывает сама себя, непосредственно (*прямая рекурсия*) или через другие программы (*косвенная рекурсия*).

Определение функции fakt в программе содержит вызов этой же функции и поэтому является (прямо) рекурсивным.

Если некоторая программа A вызывает программу B, которая вызывает программу C, содержащую обращение к программе A, то эти три программы являются (косвенно) рекурсивными.

Корректное рекурсивное определение или рекурсивная подпрограмма должны иметь хотя бы одну нерекурсивную ветку, в которой нет ссылки на определяемое понятие или рекурсивного вызова. В примере с факториалом это - вариант: k! = 1 при k = 0. Отсутствие такой ветки приведет к бесконечному повторению рекурсивных ссылок или вызовов.

При входе в любую подпрограмму в памяти в сегменте стека создается новый экземпляр ее локальных переменных. К ним относятся определенные в подпрограмме переменные и ее параметры, в том числе *адрес точки возврата* в вызывающую программу (он является неявным параметром). При выходе из подпрограммы этот экземпляр уничтожается. Такой общий механизм выполнения подпрограмм позволяет реализовать и рекурсивные подпрограммы. В некоторых языках, где этот механизм реализован не полностью, например, Fortran, Basic, рекурсивные программы запрещены.

Приведем в качестве примера трассировочную таблицу выполнения вызова fakt(2) рекурсивной программы 8.1. Точку возврата этого вызова обозначим (1), а точку возврата рекурсивного вызова fakt(k-1) из программы обозначим (2).

Трассировочная таблица вычисления fakt(2):

Номер вызова	=	1	2	3	2	1
Точка возврата	=	(1)	(2)	(2)	(2)	(1)
K	=	2	1	0	1	2
k > 0	=	да	да	Нет		
fakt (k)	=			1	1	2

После завершения вызова 3 и возврата в точку 2 продолжается выполнение вызова 2, при возврате из него в точку 2 продолжается выполнение вызова 1. При каждом возврате восстанавливаются значения переменных вызывающей программы (предыдущего вызова).

Из трассировочной таблицы видно, что при исполнении рекурсивной программы за счет ее повторных вызовов выполняются повторяющиеся действия. Таким образом, рекурсия является еще одним способом, наряду с итерацией (циклом), для представления в алгоритме повторяющихся действий. Цикл всегда можно заменить рекурсией и, наоборот, рекурсию можно реализовать с помощью цикла.

В качестве примера рассмотрим цикл с предусловием (где S – некоторый оператор)

while (Условие) S (a)

Цикл (a) описывает алгоритмический процесс вида:

Условие? - да
S

```
Условие? - да                                    (б)
S
        . . .
Условие? - нет
```

Такой же алгоритмический процесс возникает при выполнении вызова p(); в следующей рекурсивной программе, не содержащей циклов:

```
void p()
{  if (Условие)  {
       S;   p();                                 (в)
   }
}
  . . .
p();
```

Программа (в) выполняется медленнее программы (а) за счет потерь времени на переходы к подпрограмме и возвраты. Как правило, рекурсивная программа требует больше времени и памяти, чем итеративная программа решения той же задачи, хотя и может быть короче ее. Поэтому рекурсию следует использовать в тех случаях, когда трудно составить циклический алгоритм решения задачи.

Пример 2. Составить подпрограмму вывода целого числа со знаком.

Программа 8.2 (рекурсивная)

```
/*                  Вывод целого числа x                          */
void vyvod_chisla (int x)
{   if (x < 0)
    {  x = -x;
       putchar ('-');
    }
    if (x > 9)
       vyvod_chisla (x / 10);                 /* Точка возврата 1 */
    putchar (x % 10 + '0');
}
```

В программе 8.2 используется функция putchar (**int** k); - вывод символа с кодом k в стандартный выходной файл (обычно, на экран). Аналогичный вывод выполняет printf ("%c", k);

Перед выводом отрицательного числа выводится знак «-», и знак числа изменяется на противоположный. Затем выводится абсолютное значение числа.

Оператор putchar (x % 10 + '0'); выводит младшую цифру числа x, имеющую значение x%10. Выражение x%10+ '0' равно коду цифры, имеющей числовое значение x % 10. Если x содержит более одной цифры, то

предварительно с помощью рекурсивного вызова vyvod_chisla (x / 10); выводится значение старших разрядов числа x, равное x / 10.

Например, если x = 923, то рекурсивный вызов выводит значение старших разрядов числа x, равное x/10 = 92, а затем выводится младшая цифра x, равная x % 10 = 3.

Тест. Вызов: vyvod_chisla(-923); /* Точка возврата 2 */
Выход (на экране):-923

Трассировочная таблица

Номер вызова =	1	2	3	2	1
Точка возврата =	2	1	1	1	2
x =	-923	923	92	9	92	923
(x < 0) =	да		нет	нет		
(x > 9) =		да	да	нет		

Вывод: - 9 2 3 Результат: -923

8.3. Многоразрядные числа

В некоторых задачах возникает необходимость точного выполнения арифметических операций с многоразрядными числами. Числа с количеством разрядов, превышающим возможности стандартных представлений в компьютере, иногда называют «*длинными*». Рассмотрим кратко идеи одной из возможных реализаций «длинной арифметики» неотрицательных целых чисел).

На рис. 8.3 показано сложение длинных чисел Z = X + Y. Длинное число представлено целочисленным массивом в системе счисления с основанием, равным степени числа 10: OSN = 10000 = 10^4. Это сделано для удобства ввода и вывода чисел в десятичной системе счисления. Таким образом, в данном случае каждый разряд числа соответствует четырем десятичным цифрам.

Значение каждого разряда находится в диапазоне от 0 до OSN-1 = 9999. Для этого необходимо, чтобы элемент массива занимал не менее двух байтов.

В нулевом элементе массива хранится количество разрядов числа. В остальных элементах массива с индексами от 1 до MAXDIG-1 размещаются разряды числа, пронумерованные от младших к старшим.

Сложение выполняется «столбиком» от младших разрядов к старшим, как учат в школе, и оформлено в виде подпрограммы Add 8.3.

		Разряды чисел					Число разрядов
Индекс элементов массивов j		5	4	3	2	1	0
Разряды слагаемого	X[j]		9999	9800	0051	7396	4
Разряды слагаемого	Y[j]			723	0040	8402	3
порядок сложения разрядов ←							
Перенос		1	1	0	1	0	
Сумма в разряде	sum	1	10000	10523	92	15798	
Разряды суммы	Z[j]	1	0000	0523	0092	5798	5

Рис. 8.3. Сложение длинных чисел
9999980000517396 + 72300408402 = 10000052300925798

Программа 8.3. Подпрограмма сложения длинных чисел

```
#define MAXDIG 101       // Максимальное количество разрядов + 1
#define OSN  10000       // Основание системы счисления//
Сложение длинных чисел: z := x + y; (z может совпадать с x,y)
void Add (int z[MAXDIG], int x[MAXDIG], int y[MAXDIG])
{  int j, n;
   long sum;   sum = 0;
   if (x[0] > y[0]) n=x[0] else n=y[0];
   for (j=1; j<=n; j++)
   {  if (j<=x[0])  sum=sum+x[j];
      if (j<=y[0])  sum=sum+y[j];
      if (sum < OSN) { z[j] = sum;         sum = 0; }
      else           { z[j] = sum - OSN;   sum = 1; }
   }
   if (sum > 0) { z[0] = n+1; z[n+1] = 1; }
        else z[0] = n;
}
```

Обратите внимание, что подпрограмма Add составлена так, чтобы параметр Z мог совпадать с параметром X или Y, т. е. результат можно было бы поместить на место любого слагаемого. Это делает подпрограмму более универсальной и удобной, а также способствует предотвращению возможных ошибок в ее использовании. Вообще-то, любую программу желательно писать в максимально общем виде.

Вопрос 1. Какие меры предприняты в подпрограмме Add, чтобы обеспечить возможность записывать сумму на место одного из слагаемых?

Вопрос 2. Будет ли правильно работать подпрограмма Add, если совпадут все три параметра: Z, X и Y?

Вывод длинного числа производится от старших разрядов к младшим, т. е. по убыванию индексов (программа 6.8). При основании OSN > 10 каждый разряд, кроме старшего, должен выводиться в виде нескольких десятичных цифр. Поэтому, если его значение меньше OSN / 10, то перед ним выводится недостающее число нулей.

Программа 8.4 Подпрограмма вывода длинного числа

```
//                  Вывод длинного числа x
void WriteLong (int x[MAXDIG])
{  int  j, k;
   printf(”%d”, x[x[0]]);
   for (j=x[0]-1;  j>0;  j--)
   {  k = OSN/10;
      while (k > 1 && x[j] < k)
      { putchar(’0’);
        k=k/10;
      }      printf(”%d”, x[j]]);   }
}
```

При вводе каждой цифры длиного числа X его значение необходимо сдвигать влево на одну десятичную цифру (программа 8.5). Этот сдвиг удобно выполнять умножением на 10. Введенная цифра станет младшей цифрой разряда X[1], а старшая десятичная цифра разряда X[j] становится младшей цифрой разряда X[j+1].

Программа 8.5. Подпрограмма ввода длинного числа

```
//                  Ввод длинного числа x
void ReadLong2 (int x[MAXDIG])
{  char c;                          // входной символ
   long d;                          // младшая цифра разряда
   int  j;
   do     c = getchar();            // пропуск символов до числа
   while (c =='0' || c == '\n');
   x[0] = 1;  x[1] = 0;             // x = 0;
   while (c >= '0' && c <= '9')     // c - цифра
   {  d = c-'0';
      for(j=1; j<=x[0]; j++)
      {  d = 10 * (long)x[j] + d;
         x[j] = d % OSN;
         d = d / OSN;
      }
      if(d > 0)   {  x[0]++;  x[x[0]] = d; }
      c = getchar();
   }
}
```

Вопрос 3. Какие значения может принимать основание системы счисления - константа OSN, чтобы программы 8.3 – 8.5 работали корректно?

Упражнения и задачи

1. Составить трассировочные таблицы вычисления fakt(3) по рекурсивной программе;

2. Для многоразрядных (длинных) чисел составить подпрограммы реализации следующих операций.

а) сравнение двух длинных чисел: X=Y, X<Y, X>Y, X≤Y, X≥Y;

б) умножение длинного числа на короткое (двухбайтовое);

в) умножение двух длинных чисел;

г) вычитание двух длинных чисел X-Y (для X≥Y);

д) целочисленное деление с остатком двух длинных чисел («столбиком», как учат в школе; очередную цифру частного лучше подбирать методом дихотомии);

е) преобразование длинного числа в число типа long и обратно;

ж) извлечение квадратного корня из длинного числа («столбиком»);

з) получение наибольшего общего делителя двух длинных чисел.

3. Составить программу вычисления числа Фибоначчи F_n, ($n \le 1000$), если $F_1 = F_2 = 1$, $F_j = F_{j-1} + F_{j-2}$ для $j > 0$.

4. Составить программу вычисления факториала n! ($n \le 1000$).