2.6. Указатели. Динамическое выделение памяти

Определение указателей

При обработке декларации любой переменной, например *double x*=1.5; компилятор выделяет для переменной участок памяти, размер которого определяется ее типом (*double* – 8 байт), и инициализирует его указанным значением (если таковое имеется). Далее все обращения в программе к переменной по имени заменяются компилятором на адрес участка памяти, в котором будет храниться значение этой переменной. Разработчик программы на языке Си имеет возможность определить собственные переменные для хранения адресов участков оперативной памяти. Такие переменные называются указателями.

Итак, указатель – это переменная, которая может содержать адрес некоторого объекта. Простейшая декларация указателя имеет формат

***тип * ID_указателя*;**

Например: int *a; double *f; char *w;

Здесь *тип* может быть любым, кроме ссылки или битового поля, причем *тип* может быть к этому моменту только декларирован, но еще не определен (следовательно, в структуре, например, может присутствовать указатель на структуру того же типа).

Символ *«звездочка»* относится непосредственно к *ID* указателя, поэтому для того, чтобы декларировать несколько указателей, ее нужно записывать перед именем каждого из них.

Например, в декларации:

```
int *a, *b, c;
```

определены два указателя на участки памяти для целочисленных данных, а также обычная целочисленная переменная *c*.

Значение указателя равно первому байту участка памяти, на который он ссылается.

Указатели предназначены для хранения адресов областей памяти. В языке Си имеются три вида указателей – указатели на объект известного типа, указатель типа *void* и указатель на функцию. Эти три вида различаются как своими свойствами, так и набором допустимых операций. Указатель не является самостоятельным типом данных, так как всегда связан с каким-либо конкретным типом, т.е. указатель на объект содержит адрес области памяти, в которой хранятся данные определенного типа.

Указатель типа ***void*** применяется в тех случаях, когда конкретный тип объекта, адрес которого требуется хранить, не определен (например, если в одной и той же переменной в разные моменты времени требуется хранить адреса объектов различных типов).

Указателю типа *void* можно присвоить значение указателя любого типа, а также сравнивать его с любыми другими указателями, но перед выполнением

каких-либо действий с участком памяти, на которую он ссылается, требуется явно преобразовать его к конкретному типу.

Указатель может быть константой или переменной, а также указывать на константу или переменную.

С указателями-переменными связаны две унарные операции **&** и *****.

Операция **&** означает «***взять адрес***» операнда. Операция ***** имеет смысл – «***значение, расположенное по указанному адресу***» (операция разадресации).

Таким образом, обращение к объектам любого типа как операндам операций в языке Си может производиться:

– по имени (идентификатору);

– по указателю (операция косвенной адресации):

*ID*_указателя=&*ID*_объекта; – операция разыменования;

**ID*_указателя – операция косвенной адресации.

Говорят, что использование указателя означает отказ от именования адресуемого им объекта.

Операция разадресации, или разыменования, предназначена для доступа к величине, адрес которой хранится в указателе. Эту операцию можно использовать как для получения, так и для изменения значения величины (если она не объявлена как константа).

Унарная операция получения адреса & применима к переменным, имеющим имя (*ID*), для которых выделены участки оперативной памяти. Таким образом, нельзя получить адрес скалярного выражения, неименованной константы или регистровой переменной (типа *register*).

Отказ от именования объектов при наличии возможности доступа по указателю приближает язык Си по гибкости отображения «объект – память» к языку ассемблера.

***Пример* 1**:

```
int x,      – переменная типа int ;
*y;         – указатель на объект типа int;
y = &x;     – y – адрес переменной x;
*y=1;       – косвенная адресация указателем поля x, т.е. по
              указанному адресу записать 1: x = 1.
```

***Пример* 2**:

```
int i, j = 8, k = 5, *y;
y=&i;
*y=2;       –  i = 2
y=&j;
*y+=i;      –  j += i  →  j = j+i  →  j = j + 2 = 10
y=&k;
k+=*y;      –  k += k → k = k + k = 10
(*y)++;     –  k++   → k = k + 1 = 10 + 1 = 11
```

Как видно из приведенных примеров, конструкцию **ID_указателя* можно использовать в левой части оператора присваивания, так как она является *L*-значением (см. разд. 4.3), т.е. определяет адрес участка памяти. Эту конструкцию

часто считают именем переменной, на которую ссылается указатель. С ней допустимы все действия, определенные для величин соответствующего типа (если указатель инициализирован).

***Пример* 3**:

```
int i1;                    – целая переменная;
const int i2=1;            – целая константа;
int * pi1;                 – указатель на целую переменную;
const int * pi2;           – указатель на целую константу;
int * const pi1=&i1;       – указатель-константа на целую переменную;
const int * const pi2=&i2; – указатель-константа на целую константу.
```

Как видно из примеров, модификатор *const*, находящийся между *ID* указателя и символом «*звездочка*», относится к самому указателю и запрещает его изменение, а *const* слева от звездочки задает константное значение объекта, на который он указывает. Для инициализации указателей использована операция получения адреса *&*.

Указатель подчиняется общим правилам определения области действия, видимости и времени жизни.

Операция sizeof

Формат записи:

sizeof (*параметр*);

параметр – тип или идентификатор объекта (но не *ID* функции).

Данная операция позволяет определить размер указанного параметра в байтах (тип результата *int*).

Если указан идентификатор сложного объекта (массив, структура, объединение), то результатом будет размер всего объекта. Например:

```
sizeof(int)   – результат  2(4) байта;
double b[5];
sizeof(b)     – результат 8 байт * 5 = 40 байт.
```

Динамическое распределение оперативной памяти (ОП) связано с операциями порождения и уничтожения объектов по запросу программы, при котором захват и освобождение памяти производится программно, т.е. в процессе работы программы. При этом в языке Си порождение объектов (захват памяти) и уничтожение объектов (освобождение памяти) выполняются при помощи библиотечных функций.

Инициализация указателей

При декларации указателя желательно выполнить его инициализацию, т.е. присвоение начального значения. Наиболее распространенная из ошибок в программах – непреднамеренное использование неинициализированных указателей.

Инициализатор записывается после *ID* указателя либо в круглых скобках, либо после знака равенства.

Существуют следующие способы инициализации указателя:

1. Присваивание указателю адреса существующего объекта:

а) используя операцию получения адреса переменной:

```
int a = 5;
int *p = &a;        – указателю p присвоили адрес объекта a;
int *p(&a);         – то же самое другим способом;
```

б) с помощью значения другого инициализированного указателя:

```
int *g = p;
```

Указателю-переменной можно присвоить значение другого указателя либо выражения типа указатель с использованием при необходимости операции приведения типа. Приведение типа необязательно, если один из указателей имеет тип *void **, например

```
int i,*x;
char *y;
x = &i;                    // x – поле объекта int;
y = (char *)x;             // y – поле объекта char;
y = (char *)&i;            // y – поле объекта char;
```

в) с помощью идентификаторов массива или функции, которые трактуются как адрес начала участка памяти, в котором размещается указанный объект. Причем следует учитывать тот факт, что *ID* массивов и функций являются константными указателями. Такую константу можно присвоить переменной типа указатель, но нельзя подвергать преобразованиям, например:

```
int x[100], *y;
y = x;          – присваивание константы переменной;
x = y;          – ошибка, т.к. в левой части указатель-константа.
```

2. Присваивание пустого значения:

```
int *x1 = NULL;
int *x2 = 0;
```

В первой строке используется константа *NULL*, определенная как указатель, равный нулю. Рекомендуется использовать просто цифру 0, так как это значение типа *int* будет правильно преобразовано стандартными способами в соответствии с контекстом. А так как объекта с нулевым (фиктивным) адресом не существует, пустой указатель обычно используют для контроля, ссылается указатель на конкретный объект или нет.

3. Присваивание указателю адреса выделенного участка динамической памяти:

а) с помощью операции *new* (см. разд. 16.4):

```
int *n = new int;
```

```
int *m = new int (10);
```

б) с помощью функции *malloc* (см. разд. 10.9):

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

Присваивание без явного приведения типов допускается в двух случаях:

– указателям типа *void**;

– если тип указателей справа и слева от операции присваивания один и тот же.

Если переменная-указатель выходит из области своего действия, отведенная под нее память освобождается. Следовательно, динамическая переменная, на которую ссылался указатель, становится недоступной. При этом память, на которую указывала сама динамическая переменная, не освобождается. Такая ситуация называется «замусоривание оперативной памяти». Еще одна причина появления «мусора» – когда инициализированному указателю присваивается значение другого указателя. При этом старое значение указателя теряется.

Операции над указателями

Помимо уже рассмотренных операций, с указателями можно выполнять арифметические операции сложения, инкремента (++), вычитания, декремента (--) и операции сравнения.

Арифметические операции с указателями автоматически учитывают размер типа величин, адресуемых указателями. Эти операции применимы только к указателям одного типа и имеют смысл в основном при работе со структурами данных, последовательно размещенными в памяти, например с массивами.

Инкремент перемещает указатель к следующему элементу массива, декремент – к предыдущему.

Указатель, таким образом, может использоваться в выражениях вида

$$\boldsymbol{p}\ \#\ \boldsymbol{iv}, \quad \#\#\ \boldsymbol{p}, \quad \boldsymbol{p}\ \#\#, \quad \boldsymbol{p}\ \#\ =\ \boldsymbol{iv},$$

p – указатель, *iv* – целочисленное выражение, # – символ операции '+' или '–'.

Результатом таких выражений является увеличенное или уменьшенное значение указателя на величину *iv* * *sizeof*(**p*), т.е. если указатель на определенный тип увеличивается или уменьшается на константу, его значение изменяется на величину этой константы, умноженную на размер объекта данного типа.

Текущее значение указателя всегда ссылается на позицию некоторого объекта в памяти с учетом правил выравнивания для соответствующего типа данных. Таким образом, значение *p # iv* указывает на объект того же типа, расположенный в памяти со смещением на *iv* позиций.

При сравнении указателей могут использоваться отношения любого вида («>», «<» и т.д.), но наиболее важными видами проверок являются отношения равенства и неравенства («==», «!=»).

Отношения порядка имеют смысл только для указателей на последовательно размещенные объекты (элементы одного массива).

Разность двух указателей дает число объектов адресуемого ими типа в соответствующем диапазоне адресов, т.е. в применении к массивам разность указателей, например, на третий и шестой элементы равна 3.

Очевидно, что уменьшаемый и вычитаемый указатели должны принадлежать одному массиву, иначе результат операции не имеет практической ценности и может привести к непредсказуемому результату. То же можно сказать и о суммировании указателей.

Значение указателя можно вывести на экран с помощью функции *printf*, используя спецификацию %***p*** (*pointer*), результат выводится в шестнадцатеричном виде.

Рассмотрим фрагмент программы:

```
int a = 5, *p, *p1, *p2;
p = &a;
p2 = p1 = p;
++p1;
p2 += 2;
printf("a = %d , p = %d , p = %p , p1 = %p , p2 = %p .\n", a, *p, p, p1, p2);
```

Результат может быть следующим:

*a = 5 , *p = 5 , p = FFF4 , p1 = FFF6, p2 = FFF8 .*

Графически это выглядит следующим образом (в 16-разрядном процессоре на тип *int* отводится 2 байта):

	FFF5	FFF7	FFF9	
FFF4	FFF6	FFF8	FFF10	
p	*p*1	*p*2		

p = FFF4,

*p*1 = FFF6 = (FFF4 + 1*sizeof(*p)) → FFF4 + 2 (*int*)

*p*2 = FFF8 = (FFF4 + 2*sizeof(*p)) → FFF4 + 2*2

На одну и ту же область памяти (как видно из приведенного примера), может ссылаться несколько указателей различного типа. Но примененная к ним операция разадресации даст разные результаты.

При смешивании в выражении указателей разных типов явное преобразование типов требуется для всех указателей, кроме *void**.

Явное приведение типов указателей позволяет получить адрес объекта любого типа:

```
type *p;
p = (type*) &object;
```

Значение указателя *p* позволяет работать с переменной *object* как объектом типа *type*.

Указатели на указатели

Указатели, как и переменные любого другого типа, могут объединяться в массивы.

Объявление ***массива указателей*** на целые числа имеет вид

```
int *a[10], y;
```

Теперь каждому из элементов массива указателей *a* можно присвоить адрес целочисленной переменной *y*, например: *a*[1]=&*y*;

Чтобы теперь найти значение переменной *y* через данный элемент массива *a*, необходимо записать **a*[1].

В языке Си можно описать переменную типа «***указатель на указатель***». Это ячейка оперативной памяти (переменная), в которой будет храниться адрес указателя на некоторую переменную. Признак такого типа данных – повторение символа «*» перед идентификатором переменной. Количество символов «*» определяет уровень вложенности указателей друг в друга. При объявлении указателей на указатели возможна их одновременная инициализация. Например:

```
int a=5;
int *p1=&a;
int **pp1=&p1;
int ***ppp1=&pp1;
```

Если присвоить переменной *a* новое значение, например 10, то одинаковые результаты будут получены в следующих операциях:

```
a=10;   *p1=10;  **pp1=10;  ***ppp1=10;
```

Для доступа к области ОП, отведенной под переменную *a*, можно использовать и индексы. Эквивалентны следующие выражения:

```
*p1       ~    p1[0] ;
**pp1     ~    pp1[0][0] ;
***ppp1   ~    ppp1[0][0][0] .
```

Фактически, используя указатели на указатели, мы имеем дело с многомерными массивами.

Работа с динамической памятью

Указатели чаще всего используют при работе с динамической памятью, которую иногда называют «***куча***» (перевод английского слова *heap*). Это свободная память, в которой можно во время выполнения программы выделять место в соответствии с потребностями. Доступ к выделенным участкам динамической памяти производится только через указатели. Время жизни динамических объектов – от точки создания до конца программы или до явного освобождения памяти.

С другой стороны, некоторые задачи исключают использование структур данных фиксированного размера и требуют введения структур динамических, способных увеличивать или уменьшать свой размер уже в процессе работы программы. Основу таких структур составляют динамические переменные.

Динамическая переменная хранится в некоторой области ОП, не обозначенной именем, и обращение к ней производится через переменную-указатель.

Но вначале рассмотрим еще одну операцию языка Си, основное назначение которой – работа с участками оперативной памяти.

Библиотечные функции

Функции для манипулирования динамической памятью в стандарте Си следующие:

void ***calloc**(*unsigned* **n**, *unsigned* **size**); – выделение памяти для размещения **n** объектов размером **size** байт и заполнение полученной области нулями; возвращает указатель на выделенную область памяти;

void ***malloc** (*unsigned* **n**) – выделение области памяти для размещения блока размером **n** байт; возвращает указатель на выделенную область памяти;

void ***realloc** (*void* ***b**, *unsigned* **n**) – изменение размера размещенного по адресу **b** блока на новое значение **n** и копирование (при необходимости) содержимого блока; возвращает указатель на перераспределенную область памяти; при возникновении ошибки, например, нехватке памяти, эти функции возвращают значение **NULL**, что означает отсутствие адреса (нулевой адрес);

coreleft (*void*) – получение размера свободной памяти в байтах только для *MS DOS* (используется в *Borland* C++), тип результата: *unsigned* – для моделей памяти *tiny*, *small* и *medium*; *unsigned long* – для моделей памяти *compact*, *large* и *huge*;

void ***free*** (*void* ***b**) – освобождение блока памяти, адресуемого указателем **b**.

Для использования этих функций требуется подключить к программе в зависимости от среды программирования заголовочный файл ***alloc.h*** или ***malloc.h***.

В языке C++ введены операции захвата и освобождения памяти *new* и *delete*, рассматриваемые в разд. 16.4.

Связь указателей и массивов

Идентификатор одномерного массива – это адрес памяти, начиная с которого он расположен, т.е. адрес его первого элемента. Таким образом, работа с массивами тесно взаимосвязана с применением указателей. Рассмотрим связь указателей с элементами одномерного массива на примере.

Пусть объявлены одномерный целочисленный массив *a* из 5 элементов и указатель *p* на целочисленные переменные:

```
int a[5]={1, 2, 3, 4, 5}, *p;
```

ID массива *a* является константным указателем на его начало, т.е. *a* = &*a*[0] – адрес начала массива. Расположение массива *a* в оперативной памяти, выделенной компилятором, может выглядеть следующим образом:

a[0]	*a*[1]	*a*[2]	*a*[3]	*a*[4]	– элементы массива;
1	2	3	4	5	– значения элементов массива;
4000	4002	4004	4006	4008	– символические адреса.

Указатель *a* содержит адрес начала массива и в нашем примере равен 4000 (*a* = = 4000).

Если установить указатель *p* на объект *a*, т.е. присвоить переменной-указателю адрес первого элемента массива:

p = *a*;

что эквивалентно выражению *p* = &*a*[0]; то получим, что и *p* = 4000. Тогда с учетом адресной арифметики обращение к *i*-му элементу массива ***a*** может быть записано следующими выражениями:

a[*i*] ~ *(*a*+*i*) ~ *(*p*+*i*) ~ *p*[*i*] ,

приводящими к одинаковому результату.

Идентификаторы *a* и *p* – указатели, очевидно, что выражения *a*[*i*] и *(*a*+*i*) эквивалентны. Отсюда следует, что операция обращения к элементу массива по индексу применима и при его именовании переменной-указателем. Таким образом, для любых указателей можно использовать две эквивалентные формы выражений для доступа к элементам массива: *p*[*i*] и *(*p*+*i*). Первая форма удобнее для читаемости текста, вторая – эффективнее по быстродействию программы.

Например, для получения значения 4-го элемента массива можно написать *a*[3] или *(*a*+3), результат будет один и тот же, а операции *a*[3] = 8 и *(*p*+3) = 8 дадут тождественный результат, т.к. *p*+3 = 4000+3**sizeof*(*int*) = = 4000+3*2 = 4006, т.е. указатель *p* установлен на четвертый по счету элемент массива.

Очевидна и эквивалентность выражений:

– получение адреса начала массива в ОП:

&*a*[0] ↔ &(**a*) ↔ *a*

– обращение к первому элементу массива:

**a* ↔ *a*[0]

Последнее утверждение объясняет правильность выражения для получения количества элементов массива:

```
int x[]={1, 2, 3, 4, 5, 6, 7};
```

Размер *n* должен быть целочисленной константой:

```
int n = sizeof(x) / sizeof(*x);
```

Пример создания одномерного динамического массива

В языке Си размерность массива при объявлении должна задаваться константным выражением.

Если до выполнения программы неизвестно, сколько понадобится элементов массива, нужно использовать динамические массивы, т.е. при необходимости работы с массивами переменной размерности вместо массива достаточно объявить указатель требуемого типа и присвоить ему адрес свободной области памяти (захватить память).

Память под такие массивы выделяется с помощью функций ***malloc*** и ***calloc*** (операцией ***new*)** во время выполнения программы. Адрес начала массива хранится в переменной-указателе. Например:

```
int n = 10;
double *b = (double *) malloc(n * sizeof (double));
```

В примере значение переменной *n* задано, но может быть получено и программным путем.

Обнуления памяти при ее выделении не происходит. Инициализировать динамический массив при декларации нельзя.

Обращение к элементу динамического массива осуществляется так же, как и к элементу обычного – например *a*[3]. Можно обратиться к элементу массива и через косвенную адресацию – *(*a* + 3). В любом случае происходят те же действия, которые выполняются при обращении к элементу массива, декларированного обычным образом.

После работы захваченную под динамический массив память необходимо освободить, для нашего примера *free*(*b*);

Таким образом, время жизни динамического массива, как и любой динамической переменной – с момента выделения памяти до момента ее освобождения. Область действия элементов массива зависит от места декларации указателя, через который производится работа с его элементами. Область действия и время жизни указателей подчиняются общим правилам для остальных объектов программы.

Пример работы с динамическим массивом:

```
#include <alloc.h>
void main()
{
        double *x;
        int n;
        printf("\nВведите размер массива – ");
        scanf("%d", &n);
        if ((x = (double*)calloc(n, sizeof(*x)))==NULL)  {        // Захват памяти
                puts("Ошибка ");
                return;
        }
            ...
// Работа с элементами массива
            ...
        free(x);                                                  // Освобождение памяти
}
```

Примеры создания одномерного и двухмерного динамических массивов с использованием операций *new* и *delete* можно посмотреть в разд. 16.4.

Адресная функция

Векторная память поддерживается почти всеми языками высокого уровня и предназначена для хранения массивов различной размерности и различных размеров. Каждому массиву выделяется непрерывный участок памяти указанного размера. При этом элементы, например, двухмерного массива X размерностью $n1 \times n2$ размещаются в ОП в следующей последовательности:

$X(0,0)$, $X(0,1)$, $X(0,2)$,... $X(0, n2–1)$, ..., $X(1,0)$, $X(1,1)$, $X(1,2)$,... $X(1, n2–1)$, ..., $X(n1–1,0)$, $X(n1–1,1)$, $X(n1–1,2)$, ..., $X(n1–1, n2–1)$.

Адресация элементов массива определяется некоторой адресной функцией, связывающей адрес и индексы элемента.

Пример адресной функции для массива X:

$$K(i, j) = n2*i + j;$$

где $i = 0,1,2,... ,(n1–1)$; $j = 0,1,2,... ,(n2–1)$; j – изменяется в первую очередь.

Адресная функция двухмерного массива $A(n,m)$ будет выглядеть так:

$$N1 = K(i, j) = m*i + j,$$

$i=0,1,..., n–1$; $j=0,1,... , m–1$.

Тогда справедливо следующее:

$$A(i, j) \leftrightarrow B(K(i, j)) = B(N1),$$

B – одномерный массив с размером $N1 = n*m$.

Например, для двухмерного массива $A(2,3)$ имеем:

(0,0)	(0,1)	(0,2)	(1,0)	(1,1)	(1,2)	– индексы массива A;
0	1	2	3	4	5	– индексы массива B.

Проведем расчеты:

$i = 0, j = 0$ $N1 = 3*0+0 = 0$ $B(0)$
$i = 0, j = 1$ $N1 = 3*0+1 = 1$ $B(1)$
$i = 0, j = 2$ $N1 = 3*0+2 = 2$ $B(2)$
$i = 1, j = 0$ $N1 = 3*1+0 = 3$ $B(3)$
$i = 1, j = 1$ $N1 = 3*1+1 = 4$ $B(4)$
$i = 1, j = 2$ $N1 = 3*1+2 = 5$ $B(5)$

Аналогично получаем адресную функцию для трехмерного массива $X(n1, n2, n3)$:

$$K(i, j, k) = n3*n2*i + n2*j + k ,$$

где i = 0,1,2,... ,($n1$–1); j = 0,1,2,... ,($n2$–1);); k = 0,1,2,... ,($n3$–1); значение k – изменяется в первую очередь.

Для размещения такого массива потребуется участок ОП размером ($n1$*$n2$*$n3$)**sizeof*(*type*). Рассматривая такую область как одномерный массив Y(0,1,..., $n1$*$n2$*$n3$), можно установить соответствие между элементом трехмерного массива X и элементом одномерного массива Y:

$$X(i, j, k) \leftrightarrow Y(K(i, j, k)) .$$

Необходимость введения адресных функций возникает лишь в случаях, когда требуется изменить способ отображения с учетом особенностей конкретной задачи.

Пример создания двухмерного динамического массива

Напомним, что *ID* двухмерного массива – указатель на указатель. На рис. 10.1 приведена схема расположения элементов, причем в данном случае сначала выделяется память на указатели, расположенные последовательно друг за другом, а затем каждому из них выделяется соответствующий участок памяти под элементы.

```
        . . .
    int  **m, n1, n2, i, j;
    puts(" Введите размеры массива (строк, столбцов): ");
    scanf("%d%d", &n1, &n2);
// Захват памяти для указателей – A (n1=3)
    m = (int**)calloc(n1, sizeof(int*));
    for (i=0; i<n1; i++)              // Захват памяти для элементов – B (n2=4)
        *(m+i) = (int*)calloc(n2, sizeof(int));
    for ( i=0; i<n1; i++)
        for ( j=0; j<n2; j++)
                m[i] [j] = i+j;          //  *(*(m+i)+j) = i+j;
        . . .
    for(i=0; i<n; i++) free(m[i]);       // Освобождение памяти
    free(m);
        . . .
```