6. Структуры

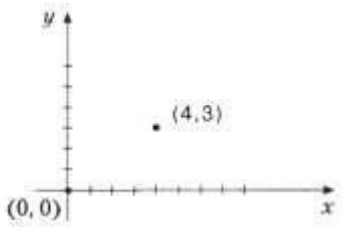
Структура — это одна или несколько переменных (возможно, различных типов), которые для удобства работы с ними сгруппированы под одним именем. (В некоторых языках, в частности в Паскале, структуры называются записями.) Структуры помогают в организации сложных данных (особенно в больших программах), поскольку позволяют группу связанных между собой переменных трактовать не как множество отдельных элементов, а как единое целое.

Традиционный пример структуры — строка платежной ведомости. Она содержит такие сведения о служащем, как его полное имя, адрес, номер карточки социального страхования, зарплата и т. д. Некоторые из этих характеристик сами могут быть структурами: например, полное имя состоит из нескольких компонент (фамилии, имени и отчества); аналогично адрес, и даже зарплата. Другой пример (более типичный для Си) — из области графики: точка есть пара координат, прямоугольник есть пара точек и т. д.

Главные изменения, внесенные стандартом ANSI в отношении структур, — это введение для них операции присваивания. Структуры могут копироваться, над ними могут выполняться операции присваивания, их можно передавать функциям в качестве аргументов, а функции могут возвращать их в качестве результатов. В большинстве компиляторов уже давно реализованы эти возможности, но теперь они точно оговорены стандартом. Для автоматических структур и массивов теперь также допускается инициализация.

6.1. Основные сведения о структурах

Сконструируем несколько графических структур. В качестве основного объекта выступает точка с координатами x и y целого типа.



Указанные две компоненты можно поместить в структуру, объявленную, например, следующим образом:

```
struct point
    { int x;
    int y;
};
```

Объявление структуры начинается с ключевого слова `struct` и содержит список объявлений, заключенный в фигурные скобки. За словом `struct` может следовать имя, называемое *тегом*[9] *структуры*, (`point` в нашем случае). Тег дает название структуре данного вида и далее может служить кратким обозначением той части объявления, которая заключена в фигурные скобки.

Перечисленные в структуре переменные называются *элементами* (*members*)[10]. Имена элементов и тегов без каких-либо коллизий могут совпадать с именами обычных переменных (т. е. не элементов), так как они всегда различимы по контексту. Более того, одни и те же имена элементов могут встречаться в разных структурах, хотя, если следовать хорошему стилю программирования, лучше одинаковые имена давать только близким по смыслу объектам.

---

[9] От английского слова *tag* — ярлык, этикетка. — *Примеч. пер.*

[10] В некоторых изданиях (в том числе во 2-м издании на русском языке этой книги) *structure members* переводится как *члены структуры*. — *Примеч. ред.*

Объявление структуры определяет тип. За правой фигурной скобкой, закрывающей список элементов, могут следовать переменные точно так же, как они могут быть указаны после названия любого базового типа. Таким образом, выражение

```
struct {…} x, y, z;
```

с точки зрения синтаксиса аналогично выражению

```
int x, y, z;
```

в том смысле, что и то и другое объявляет `x`, `y` и `z` переменными указанного типа; и то и другое приведет к выделению памяти соответствующего размера.

Объявление структуры, не содержащей списка переменных, не резервирует памяти; оно просто описывает шаблон, или образец структуры. Однако если структура имеет тег, то этим тегом далее можно пользоваться при определении структурных объектов. Например, с помощью заданного выше описания структуры point строка

```
struct point pt;
```

определяет структурную переменную `pt` типа `struct point`. Структурную переменную при ее определении можно инициализировать, формируя список инициализаторов ее элементов в виде константных выражений:

```
struct point maxpt = { 320, 200 };
```

Инициализировать автоматические структуры можно также присваиванием или обращением к функции, возвращающей структуру соответствующего типа.

Доступ к отдельному элементу структуры осуществляется посредством конструкции вида:

имя-структуры.элемент

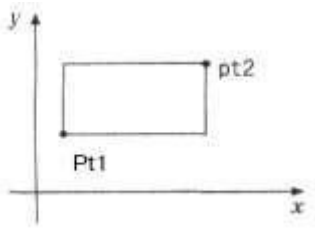
Оператор доступа к элементу структуры `.` (точка) соединяет имя структуры и имя элемента. Чтобы напечатать, например, координаты точки `pt`, годится следующее обращение к `printf`:

```
printf("%d,%d", pt.x, pt.y);
```

Другой пример: чтобы вычислить расстояние от начала координат (0,0) до `pt`, можно написать

```
double dist, sqrt(double);
dist = sqrt((double)pt.x * pt.x + (double)pt.y * pt.y);
```

Структуры могут быть вложены друг в друга. Одно из возможных представлений прямоугольника — это пара точек на углах одной из его диагоналей:




```
struct rect {
    struct point pt1;
    struct point pt2;
};
```

Структура `rect` содержит две структуры `point`. Если мы объявим `screen` как

```
struct rect screen;
```

то

```
screen.pt1.x
```

обращается к координате `x` точки `pt1` из `screen`.

6.2. Структуры и функции

Единственно возможные операции над структурами — это их копирование, присваивание, взятие адреса с помощью `&` и осуществление доступа к ее элементам. Копирование и присваивание также включают в себя передачу функциям аргументов и возврат ими значений. Структуры нельзя сравнивать. Инициализировать структуру можно списком константных значений ее элементов; автоматическую структуру также можно инициализировать присваиванием.

Чтобы лучше познакомиться со структурами, напишем несколько функций, манипулирующих точками и прямоугольниками. Возникает вопрос: а как передавать функциям названные объекты? Существует по крайней мере три подхода: передавать компоненты по отдельности, передавать всю структуру целиком и передавать указатель на структуру. Каждый подход имеет свои плюсы и минусы.

Первая функция, `makepoint`, получает два целых значения и возвращает структуру `point`.

```
/* makepoint: формирует точку по компонентам x и y */
struct point makepoint(int x, int y)
{
    struct point temp;
    temp.x = x; temp.y
    = y; return temp;

}
```

Заметим: никакого конфликта между именем аргумента и именем элемента структуры не возникает; более того, сходство подчеркивает родство обозначаемых им объектов.

Теперь с помощью `makepoint` можно выполнять динамическую инициализацию любой структуры или формировать структурные аргументы для той или иной функции:

```
struct rect screen;
struct point middle;
struct point makepoint(int, int);

screen.pt1 = makepoint(0, 0);
screen.pt2 = makepoint(XMAX, YMAX);
middle = makepoint((screen.pt1.x + screen. pt2.x)/2,
    (screen.pt1.y + screen.pt2.y)/2);
```

Следующий шаг состоит в определении ряда функций, реализующих различные операции над точками. В качестве примера рассмотрим следующую функцию:

```
/* addpoint: сложение двух точек */
struct point addpoint(struct point p1, struct point p2)
{
    p1.x += p2.x;
```

```
    p1.y += p2.y;
    return p1;
}
```

Здесь оба аргумента и возвращаемое значение — структуры. Мы увеличиваем компоненты прямо в `p1` и не используем для этого временной переменной, чтобы подчеркнуть, что структурные параметры передаются по значению так же, как и любые другие.

В качестве другого примера приведем функцию `ptinrect`, которая проверяет: находится ли точка внутри прямоугольника, относительно которого мы принимаем соглашение, что в него входят его левая и нижняя стороны, но не входят верхняя и правая.

```
/* ptinrect: возвращает 1, если p в r, и 0 в противном случае
*/ int ptinrect(struct point p, struct rect r)
{
    return p.x >= r.ptl.x && p.x < r.pt2.x
        && p.y >= r.ptl.y && p.y < r.pt2.y;
}
```

Здесь предполагается, что прямоугольник представлен в стандартном виде, т. е. координаты точки `pt1` меньше соответствующих координат точки `pt2`. Следующая функция гарантирует получение прямоугольника в каноническом виде.

```
#define min(a, b) ((a) < (b) ? (a) : (b))
#define max(a, b) ((a) > (b) ? (a) : (b))

/* canonrect: канонизация координат прямоугольника */
struct rect canonrect(struct rect r)
{
    struct rect temp;
    temp.pt1.x  =  min(r.pt1.x,  r.pt2.x);
    temp.ptl.y  =  min(r.pt1.y,  r.pt2.y);
    temp.pt2.x  =  max(r.pt1.x,  r.pt2.x);
    temp.pt2.y  =  max(r.pt1.y,  r.pt2.y);
    return temp;
}
```

Если функции передается большая структура, то, чем копировать ее целиком, эффективнее передать указатель на нее. Указатели на структуры ничем не отличаются от указателей на обычные переменные. Объявление

```
struct point *pp;
```

сообщает, что `pp` — это указатель на структуру типа `struct point`. Если `pp` указывает на структуру `point`, то `*pp` — это сама структура, а `(*pp).x` и `(*pp).y` — ее элементы. Используя указатель `pp`, мы могли бы написать

```
struct point origin,
*pp; pp = &origin;
printf ("origin: (%d,%d)\n", (*pp).x, (*pp).y);
```

Скобки в `(*pp).x` необходимы, поскольку приоритет оператора `.` выше, чем приоритет `*`. Выражение `*pp.x` будет проинтерпретировано как `*(pp.x)`, что неверно, поскольку `pp.x` не является указателем.

Указатели на структуры используются весьма часто, поэтому для доступа к ее элементам была придумана еще одна, более короткая форма записи. Если `p` — указатель на структуру, то

```
p -> элемент-структуры
```

есть ее отдельный элемент. (Оператор `->` состоит из знака `-`, за которым сразу следует знак `>`.) Поэтому `printf` можно переписать в виде

```
printf("origin: (%d,%d)\n", pp->x, pp->y);
```

Операторы `.` и `->` выполняются слева направо. Таким образом, при наличии объявления

```
struct rect r, *rp = &r;
```

следующие четыре выражения будут эквивалентны:

```
r.pt1.x rp-
>pt1.x
(r.pt1).x
(rp->pt1).x
```

Операторы доступа к элементам структуры `.` и `->` вместе с операторами вызова функции `()` и индексации массива `[]` занимают самое высокое положение в иерархии приоритетов и выполняются раньше любых других операторов. Например, если задано объявление

```
struct {
int len;
char *str;
} *p;
```

то

```
++p->len
```

увеличит на 1 значение элемента структуры `len`, а не указатель `p`, поскольку в этом выражении как бы неявно присутствуют скобки: `++(p->len)`. Чтобы изменить порядок выполнения операций, нужны явные скобки. Так, в `(++p)->len`, прежде чем взять значение `len`, программа прирастит указатель `p`. В `(p++)->len` указатель `p` увеличится после того, как будет взято значение `len` (в последнем случае скобки не обязательны).

По тем же правилам `*p->str` обозначает содержимое объекта, на который указывает `str`; `*p->str++` прирастит указатель `str` после получения значения объекта, на который он указывал (как и в выражении `*s++`); `(*p->str)++` увеличит значение объекта, на который указывает `str`; `*p++->str` увеличит `p` после получения того, на что указывает `str`.