ЕН.Ф.02 – Информатика и программирование

Лекция 7. Конструируемые типы данных. Указатели: теоретическое введение и практическое использование

Конова Елена Александровна
E_Konova@mail.ru

Введение

Любой объект программы (переменная, массив, функция и другие) имеют:

а) необходимые атрибуты:

имя – адресует область памяти, например, имя `Alpha` сохраняет объект по адресу **`FFF0`**,

тип – определяет механизм выделения памяти, например, **`int`** получает 4 байта.

б) операции над объектом:

взять значение (по указанному адресу),

изменить значение.

Есть еще термин «ячейка памяти», который устарел, но объясняет суть понятия «объект программы».

Определение указателя

Указатели – программные объекты, значением которых являются адреса других объектов или области памяти.

Модель памяти

Оперативная память, это поток адресуемых байт. Возможно разделение на машинные слова. Нумерация адресов в 16-ричной системе.

Адресное пространство процесса разделено на сегменты.

Адрес_объекта = сегмент + смещение (16-ричный адрес внутри сегмента)

Значением указателя может быть пустое значение, не равное никакому адресу. Он объявлено `null` в нескольких заголовочных файлах, например, `<stdef.h>` , `<stdio.h>`.

Синтаксис указателя

При объявлении переменных типа указатель, признаком того, что вводится указатель, является *. Должно быть определено имя типа, на который указатель ссылается.

```
int   * pa;      // Указатель на область, содержащую
                 // целое число.
float * px;      //    -/-    действительное число.
char  * pc;      //    -/-    символ.
void  * pv;      // Указатель на область неизвестного
                 // типа (поток байт).
```

Почему должно быть имя типа?

Указатель, это адрес первого байта и количество адресуемых байт.

Переменная типа указатель получает 4 байта (зависит от модели памяти).

Пример

```
int    x = 3;   // 4 байта для x.
int    *px;  // Объявлен указатель.
// Операция &x получит адрес переменной x.
px = &x;     // Указатель = адрес.
int    k=3;  // Другая переменная.
// Операция * получит значение по указанному адресу.
int    *pk;  // Косвенная адресация.
pk = &k;     // Получен адрес переменной k.
x = *pk;     // Переменная x получила значение
        // того объекта, который хранится по
        // адресу, имеющему значение pk.
```

Как выглядят адреса, нужно посмотреть в отладчике.

Размещение в памяти

Статические данные размещаются в стеке данных.

Операции над указателями

Операции над указателями разрешены, их можно разделить на группы:

специальные, предназначенные исключительно для работы с адресами,

обычные, расширяющие возможности программиста.

1. Операция получения адреса

& – получить адрес, унарная операция, которая:

- может быть применена к операнду любого типа,
- возвращает 16-ричный адрес объекта.

Например,

```
int    x = 3;   // x получает 4 байта.
int    * px; // px получает 4 байта.
// операция & x получит адрес переменной x.
px = & x;       // Адрес можно присвоить
указателю.
```

Замечание. Операция применяется только к именованным объектам, размещенным в памяти.

2. Операция получения значения

* – взять значение по адресу – унарная операция косвенной адресации (разыменования, раскрытия ссылки, обращения по адресу).

Операндом может быть только указатель, а возвращаемое значение:

- имеет тип той переменной, на которую показывает указатель,
- возвращает значение, размещенное в той области памяти, на которую ссылается указатель.

Пример

```
int    x;
int    k = 3;
int    *pk;
// Переменная pk получает адрес переменной k.
pk = &k;
// Переменная x получает значение переменной,
    // которая хранится по адресу pk.
x = *pk;
```

Замечание.

void может адресовать объект любого типа, но к нему нельзя применить операцию *.

3. Операция присваивания для указателей

Разрешается присваивать значения переменных одного типа или **null** (пустой адрес).

```
int  x=3;
int  *px;
int  px = null;   // px = 0;
int  *py=&x;
px = py;          // px и py адресуют ?
```

Присваивание значений разных типов допустимо, при этом происходит приведение и преобразование типов.

Для указателей механизм приведения имеет существенные особенности (пример).

Приведение и преобразование типов

Явное преобразование типа имеет синтаксис:

(имя_типа) Выражение

Например,

```
y = (int) x;
```

Неизбежна потеря данных при преобразовании от большего типа к к меньшему.

Для указателей преобразование записывается так же, но с операцией *:

```
int     * x;
double * y;
y = (int *)  x;
x = (double *) y;     // Потеря данных.
```

4. Унарные ++ и --

Унарные ++ и -- еще называют операциями смещения указателя.

Эти операции изменяют значение адреса в зависимости от типа данных, с которым связан указатель, а именно:

Для **`char`** – на 1 байт.

Для **`int`** – на 4 байта.

Для **`double`** – на 8 байт.

И так далее.

5. Сложение и вычитание

Можно прибавить к адресу целое число. Новое значение определит смещение в байтах нового адреса в зависимости от типа указателя:

число*`sizeof`(тип_указателя)

Вычитание адресов можно применить:

а) к указателю и числу,

б) или к двум указателям.

Во втором случае разность (со знаком), это расстояние в единицах, кратных размеру одного объекта указанного типа.

```
int *px, *py;
```

В байтах:

```
(px - py)*sizeof(px)
```

6. Сравнение указателей

Разрешается сравнивать только указатели одинакового типа или с **null**.

Могут использоваться все операции отношения, но чаще == или !=.

Назначение - для проверки результата выполнения операции.

Например, многие функции возвращают **null** в случае неудачного исполнения, в том числе scanf().

Приоритеты операций

1. Унарные операции косвенной адресации * и получения адреса & старше, чем все остальные.

2. Аддитивные операции.

3. Отношения.

4. ++ и -- следующим образом:

а) в префиксной форме операции ++ и -- выполняются прежде других операций;

б) в постфиксной форме операции ++ и -- выполняются после других операций.

Практическое использование указателей

Простые цели использования указателей.

1. Косвенная адресация.
2. Динамические массивы.
3. Указатели и функции.

Косвенная адресация

Указатели и массивы: синтаксис языка C++ определяет имя массива как адрес его первого элемента (с нулевым значением индекса).

Это дает возможность обращаться к элементам массива смещением адреса текущего объекта относительно начала массива.

В общем случае, управление при обработке массивов, это адресация элементов.

Прямая адресация для элементов массива имеет синтаксис:

```
for (int i=0; i<n; i++)
    {
        Arr[i]; // Обращение к i-тому элементу.
    }
```

Косвенная адресация

Обращение к элементам массива может быть выполнено не только с помощью операции [], но и с использованием операции * (косвенная адресация).

Для определения номера элемента внутри массива используется смещение указателя от первого элемента.

Пусть

```
int Arr[] = {10,20,30,40};
```

Здесь `Arr` – адрес первого элемента массива, `Arr[0]`, значит,

`Arr = &Arr[0]`

`Arr+1 = Arr[1]`, где +1 = смещение адреса на `sizeof(int)` байта.

`Arr+2 = Arr[2]`, и так далее.

Косвенная адресация

Косвенная адресация элементов массива выполняется через указатель.

```
int *pti;  // Замечание! тип - как у элементов
           // массива (pti - синоним массива).
...
for( pti = Arr; pti < Arr+N; pti ++)
   // смещение на sizeof(int) байт.
 {
    // обращение к * pti
 }
```

Важно! pti является временной (рабочей) переменной, ее значение изменяется.

Значение Arr изменяться не должно.

Массивы переменной длины

Недостаток синтаксического определения массива – длина должна быть задана константой.

Инструменты для использования массивов условно переменной длины:

1. Использование константного выражения.
2. Массивы условно переменной длины.
3. Динамические массивы.

Использование константного выражения

Определяем длину массива как **define** константу:

```
#define N 100     // Наибольшее значение длины
// Везде в управлении указываем имя этой константы
for (i=0; i<N; i++)
   {
      // Управление завязано на N
   }
```

При изменении длины, меняем N и перекомпилируем проект.

Недостатки:

а) передача в функции,

б) невозможно изменить длину при работе.

Массивы условно переменной длины

Длина массива задана константой, она заведомо большего размера, чем требуется. Для адресации используется часть выделенной памяти. Вводится переменная, обозначающая реальную длину массива, и эта переменная используется для управления.

```
#define N 100 // По максимуму
int Arr [N];  // Описание массива
int len;    // Реальная длина
// len можно ввести, присвоить, вычислить len < N.
for (i=0; i<len; i++)
 {
        // Управление завязано на len
  }
```

Вывод

При статическом распределении памяти для элементов массива есть существенный недостаток – общее количество элементов массива должно быть известно при компиляции, когда происходит распределение памяти для элементов массива.

Динамические массивы

Для статических массивов память выделяется на этапе компиляции программы в стеке данных, и ее размер не может быть изменен при работе программы.

Для динамических массивов память выделяется в процессе работы программы в области динамической памяти (куча – `heap`), и ее размер может быть изменен при работе программы.

Решается через механизм указателей.

Механизмы выделения памяти в C++

В C++ для работы с динамической памятью используются операции **new** и **delete**.

Соответственно выделяют память для объекта и разрушают объект, возвращая память в кучу (**heap**).

Синтаксис:

```
Указатель = new тип имя_объекта [количество];
delete имя_объекта;
```

Для массива:

```
delete []имя;
```

Семантика

Операция **new** пытается открыть объект с именем «имя_объекта» путем выделения **sizeof** (имя_объекта) байт в куче.

Операция **delete** удаляет объект с указанным именем, возвращая **sizeof** (имя_объекта) байт в кучу.

Объект существует от момента создания до момента разрушения или до конца программы.

Замечание. При выделении памяти переменной (массиву) через указатель, переменная (массив) не имеет собственного имени. Обращение к переменной (массиву) выполняется только посредством указателя.

Пример

```
Type  *nameptr;
// Здесь Type - любой тип кроме функции
...
if (!(nameptr = new Type))
    {
      printf("Нет места в памяти.\n");
      exit();
    }
// Использование объекта по назначению
...
delete nameptr;    // Разрушение объекта.
```

Указатели и функции

1. Передача параметров в функцию (из функции) по адресу.

Параметр по значению: при обращении к функции и передаче ей параметров создается локальная копия объекта в теле функции.

Параметр по адресу: в качестве параметра передается адрес объекта вызывающей программы (признак операции &).

В классическом С в теле функции необходимо разименовать переменную (операция *).

```
void change_5 (int * ptr)
// ptr -- указатель на переменную int.
{
   *ptr += 5;
}
```

Параметр по ссылке

В C++ признак передачи параметра по ссылке указан символом & в списке формальных параметров при описании функции.

```
void change_5 (int & ptr) // Передача по ссылке
{
    ptr += 5;
}
```

Массивы как параметры функций

Особенность массива в том, что он является указателем. При передаче массива в функцию передается адрес, следовательно, функция может изменить элементы массива.

Массив не защищен от изменения функцией.

Способы записи формальных параметров в заголовке функции

```
Тип_функции имя_функции (тип_массива имя [], int длина)
{
... // имя, это Адрес массива.
}
```

Или:

```
тип_функции имя_функции (тип_массива  * имя, int длина)
{
... // имя, это Адрес массива.
}
```

Выводы

При обращении к функции указывается только имя массива и длина.

Достоинства:

- использование массивов переменной длины,
- решение задач обработки массивов в общем виде.

Функции, возвращающие указатели

Функция может вернуть значение:

а) переменной базового типа, тогда ее тип равен одному из базовых типов;

б) переменной сложного конструируемого типа, тогда ее тип – указатель, и функция должна вернуть адрес объекта указанного типа.

Этот механизм используется во многих библиотечных функциях. Как правило, возвращает указатель на вновь созданное значение.

Пример. Функция получения массива

Динамическая память может быть выделена в любом месте программы, в том числе в **main** или в теле функции:

Например, в **main** объявлен указатель

```
int *Arr;
```

Функция может создать массив и вернуть его адрес

```
Arr = get_Arr (n);      // n = длина массива
```

Реализация функции

```
int * get_Arr (int len)
{
int * A;
A =  new int[len]; // Выделена память.
printf("Input %d elements",n);
   for (int i=0,i<n;i++)
       scanf("%d",A[i]);
return A;
// Возвращается адрес выделенной динамически
   памяти.
}
// В main адрес должен быть присвоен массиву.
```

Обращение к функции

В **main** чтобы получить значение, должен быть объявлен указатель:

```
int *mas;
```

Функция создает массив и возвращает его адрес:

```
mas = get_Arr (n);      // n = длина массива.
```

Динамическая память выделена в функции, это известно вызывающей программе.

Указатели на функции

Указатель на функцию как возвращаемое значение.

Функции отнесены к производным типам данных, значит, это не только и не столько алгоритм решения задачи, но и возвращаемое значение.