1 Занятие №4

1.1 Перечислимые типы

Язык Си имеет средства для определения перечислимых типов. В общем виде объявление перечислимых типов выглядит следующим образом:

```
enum <тег перечисления> { <список констант> };
```

например,

```
enum traffic_light { green, yellow, red };
```

Тег перечисления может отсутствовать, тогда определяется анонимное перечисление, которое не вводит новый тип, а вводит только новые перечислимые константы.

Если тег перечисления присутствует, то вновь введённый тип можно впоследствии использовать для определения новых переменных, структур, других типов. Имя перечислимого типа состоит из двух компонент: ключевого слова **enum** и тега перечисления. Они всегда должны использоваться вместе. Например,

```
enum traffic_light light = red;
```

Все теги перечислимых типов находятся в отдельном от обычных идентификаторов пространстве имён, то есть обычные идентификаторы и теги перечислимых типов существуют как бы «параллельно». Тег перечислимого типа перекрывает только другие теги. На одном лексическом уровне тег должен быть уникальным среди других тегов.

Константы перечислимого типа находятся в пространстве имён идентификаторов, то есть они взаимно перекрывают идентификаторы, и должны быть уникальны с идентификаторами на одном уровне вложенности. Пример,

```
enum a { c1, c2 };
enum b { c3 };
enum a a;

int f()
{
  enum a { c3 }; // перекрывает только тип enum a, но не переменную a.
                 // константа c3 перекрыла константу c3 типа enum b
  enum a b = c1;

  a = c3;        // доступ к глобальной a, присваивание ей
                 // локальной константы c3
}
```

Константам перечислимого типа можно указывать начальное значение. Оно должно быть константным выражением. По умолчанию первая константа получает значение 0, а последующие — значение предыдущей, увеличенное на 1. Например,

```
enum {a = 4, b, A = a, c = -4};
```

Обратите внимание, константы перечисления могут участвовать в константных выражениях, которые требуются при инициализации перечислимых констант и, например, при определении размера массива.

Перечислимый тип может иметь любой целый тип, достаточный для хранения значений всех его констант. Например, для перечисления

```
enum {a = 4, b, c};
```

компилятор может выделить тип **signed char**, а может и **int**. Узнать это можно операцией **sizeof**. Константы перечислимого типа могут использоваться везде, где требуется целое значение.

1.2 Операторы передачи управления

1.2.1 Оператор switch

Оператор выбора **switch** используется для передачи управления в зависимости от значения некоторого выражения целочисленного типа. Оператор записывается следующим образом:

```
switch (<выражение>) <оператор>
```

Где <оператор> почти всегда составной оператор, но, вообще говоря, это не обязательно.

Внутри оператора могут располагаться метки **case**, записываемые следующим образом

```
case <константное целое выражение>:
```

Оператор вычисляет выражение в заголовке оператора **switch**, и затем передаёт управление на метку **case** с соответствующим значением. В операторе **switch** не может содержаться две метки **case** с равным значением. Если соответствующая метка **case** не найдена, производится переход на метку **default:**, если она существует, и выход из оператора **switch** в противном случае.

Метки **case** и **default** могут содержаться во вложенных операторах (например, **while** или **for**), но не во вложенных операторах **switch**. Это значит, что метки **case** и **default** во вложенном операторах **switch** не действуют во внешнем операторе **switch**.

Например:

```
switch (a) {
  case 0:             // a == 0
    while (b) {
      case 1:         // a == 1
    }
  case 2:             // a == 2
    switch (c) {
      case 1:         // c == 1
      case 2:         // c == 2
      case 3:         // c == 3
      default:        // c != 1 && c != 2 && c != 3
    }
}                     // a != 0 && a != 1 && a != 2
```

Если внутри оператора **switch** встречается оператор **break;**, управление передаётся за оператор выбора. Оператор **break;** должен использоваться для выхода из оператора выбора после вычисления альтернативы, потому что в противном случае управление «провалится» на следующую метку **case**. Например,

```
switch (light) {
  case 0:
    printf("Stop\n");
    break;
  case 2:
    printf("Go\n");
    break;
  case 1:
    printf("Be␣ready\n");
  default:
    printf("Unknown␣light\n");
}
```

1.2.2 Оператор `break`

Оператор **`break;`** передаёт управление за пределы самого вложенного оператора **`do`**, **`while`**, **`for`** или **`switch`**. Оператор **`break;`** не может использоваться вне этих операторов. Например,

```
switch (x) {
  case a:
    while (1) {
      if (y) break;
    }
    break;
  default:
    ;
}
```

1.2.3 Оператор `continue`

Оператор **`continue;`** передаёт управление за оператор, составляющий тело цикла **`do`**, **`while`** или **`for`**, что означает немедленное начало следующей итерации цикла. Для оператора **`do`** произойдёт переход на вычисление управляющего выражения **`while`** в конце цикла, для оператора **`while`** произойдёт переход на вычисление управляющего выражения в начале цикла. Для оператора **`for`** произойдёт переход к вычислению третьего выражения заголовка цикла **`for`**.

1.2.4 Оператор `goto`

Оператор `goto <метка>;` передаёт управление на заданную метку. Метка должна находится в той же функции, помеченный оператор записывается `<метка>:`. Где `<метка>` — это произвольный идентификатор. Метки никак специально в функции не объявляются, и переходы на метки вперёд допустимы. Метки находятся в своём пространстве имён, отличном от идентификаторов и тегов, это значит, что имена меток могут совпадать с именами других объектов. Метки всегда локальны в пределах одной функции. Две метки с одним именем внутри одной функции не допускаются. Оператор перехода может входить в блок в обход инициализации, например:

```
if (x) goto label;
/* ... */
while (y) {
  int z = 1;

  /* ... */
label:
  /* ... */
}
```

тогда при переходе на метку `label` под переменную будет выделена память, но она *не будет инициализирована*.

1.2.5 Об использовании `goto`

Споры о необходимости, нужности и допустимости оператора **`goto`** не утихают уже более 30 лет. Концепция *структурного программирования* отрицает оператор **`goto`**, утверждая, что любая программа может (и должна) быть записана только с помощью операторов присваивания, **`if`** и **`while`**. Считается, что использование **`goto`** делает программу трудночитаемой, труднопонимаемой и трудносопровождаемой.

В языке Си все эти операторы присутствуют, более того, есть «не совсем структурные» операторы **`return`**, **`break`** и **`continue`**, однако оператор **`goto`** всё равно сохранён, хотя и в ограниченной форме, позволяя выполнять произвольные передачи управления в пределах одной функции. Почему? Дело в том, что в некоторых ситуациях (количество которых весьма ограничено) оператор **`goto`** наоборот, позволяет достичь большей ясности, чем использование структурных конструкций. К одной из таких ситуаций относится, например, выход из глубоко вложенных циклов.

Мы рассмотрим другую ситуацию, когда оператор **`goto`** используется для перехода на фрагмент функции, обрабатывающий ошибки. Многие функции на концептуальном уровне выглядят следующим образом: захват необходимых ресурсов (например, открытие файлов, выделение памяти), выполнение действий над ними, освобождение ресурсов. В случае отказа в выделении какого-либо ресурса функция должна завершить работу, вернув специальное значение, соответствующее ошибке.

Предположим, что функция называется `do_action`, и она возвращает 0 в случае успешного завершения и -1 в случае ошибки. Функции для работы необходимы три ресурса r_1, r_2 и r_3. Такая функция может выглядеть следующим образом:

```
 1 int do_action()
 2 {
 3   res1_t r1 = 0;
 4   res2_t r2 = 0;
 5   res3_t r3 = 0;
 6
 7   if (!(r1 = get_res1())) goto failure;
 8   if (!(r2 = get_res2())) goto failure;
 9   if (!(r3 = get_res3())) goto failure;
10
11   // выполнить требуемые действия
12
```

```
13    free_res3(r3);
14    free_res2(r2);
15    free_res1(r1);
16    return 0;
17
18  failure:
19    if (r3) free_res3(r3);
20    if (r2) free_res2(r2);
21    if (r1) free_res1(r1);
22    return -1;
23  }
```

Здесь `res`*i*`_t` — это некоторый тип данных, соответствующий *i*-му ресурсу. Каждый из этих типов данных имеет значение 0, означающее, что соответствующий ресурс не выделен. Функция `get_res`*i* выделяет ресурс *i*, возвращая 0, если ресурс не может быть выделен. Функция `free_res`*i* возвращает *i*-й ресурс системе.

В данном примере использование **`goto`** позволяет выделить обработку ошибок в отдельный фрагмент функции и чётко отделить её от основного кода, что повышает читабельности и понимаемость функции.

Следует заметить, что в языках более высокого уровня (например, **Си++**) эту роль берут на себя конструкторы и деструкторы и операторы обработки исключений.