1 Занятие №5

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 Выражения

1.2.1 Перечисление операций

1. Наивысший приоритет имеют постфиксные операции `++, --, [], (), ., ->`. Читаются слева направо.

`++`	постинкремент
`--`	постдекремент
`[]`	индексация массива, например `a[i]`
`()`	вызов функции, например `sin(M_PI)`
`.`	доступ к полю
`->`	доступ к полю через указатель

2. Следующая группа операций — префиксные операции. Читаются справа налево.

`++`	преинкремент
`--`	предекремент
`&`	взятие адреса
`*`	разыменование
`-`	
`+`	
`!`	логическое отрицание
`~`	побитовое отрицание
`sizeof`	вычисление размера типа
`()`	приведение типа

3. Остальные операции по мере убывания их приоритета. Читаются слева направо.

`*`	
`/`	
`%`	
`+`	
`-`	
`<<`	сдвиг влево
`>>`	сдвиг право
`<`	
`<=`	
`>`	
`>=`	
`==`	
`!=`	
`&`	побитовая операция AND
`^`	побитовая операция XOR
`|`	побитовая операция OR
`&&`	логическое «и»
`||`	логическое «или»
`?:`	условная операция, читается справа налево
`=`	присваивания, читаются справа налево
`*=`	
`/=`	
`%=`	
`+=`	
`-=`	
`<<=`	
`>>=`	
`&=`	
`^=`	
`|=`	
`,`	последовательное вычисление

Операции `&&` и `||` вычисляются по «короткой» схеме, то есть если значение всего выражения известно после вычисления первого операнда, второе выражение не вычисляется. Операцию `a && b` можно понимать так:

```
if (a) { if (b) 1; else 0; } else 0;
```

а операцию `a || b` так:

```
if (a) 1; else if (b) 1; else 0;
```

Операция `?:` тернарная, то есть имеет три операнда, например `a?b:c`. Сначала вычисляется условие `a`. Если оно истинно, тогда вычисляется `b`, а `c` не вычисляется, а если `a` ложно, тогда вычисляется `c`, а `b` не вычисляется. Эта операция читается справа налево, то есть выражение `a?b:c?d:e` эквивалентно `a?b:(c?d:e)`, выражение `a?b?c:d:e` эквивалентно `a?(b?c:d):e`.

Операция `,` («запятая») последовательно вычисляет оба аргумента. Значением операции является значение второго аргумента, значение первого теряется.

1.2.2 Преобразования типов при вычислении выражений

Перед вычислением арифметических операций транслятор выполняет два действия с аргументами: *целочисленное повышение* (integral promotion) и *балансировка*.

Целочисленное повышение. Кроме случая, когда выражение является аргументом операции **`sizeof`**, целочисленное выражение может иметь один из следующих типов:

- **`int`**
- **`unsigned int`**
- **`long`**
- **`unsigned long`**
- **`long long`**
- **`unsigned long long`**

Если выражение имеет тип, не перечисленный выше, транслятор повышает его до одного из этих типов. Если все значения исходного типа представимы также и значениями типа **`int`**, результатом повышения будет тип **`int`**. В противном случае результатом повышения будет тип **`unsigned int`**.

Таким образом, для типов **`signed char`**, **`short`** и для знаковых битовых полей результатом повышения является тип **`int`**. Для всех оставшихся целых типов предпочитается повышение в **`int`** когда это возможно, но в **`unsigned int`**, если это необходимо для сохранения значения во всех возможных случаях.

Балансировка. В случае вычисления инфиксного выражения, которое имеет два арифметических операнда, транслятор определяет тип выражения с помощью балансировки типов операндов. Для балансировки типов транслятор применяет следующие правила к повышенным типам операндов.

- Если балансируемые типы аргументов не **`unsigned int`** и **`long`**, результатом балансировки является тот повышенный тип операнда, который встречается позднее в последовательности типов **`int`**, **`unsigned int`**, **`long`**, **`unsigned long`**, **`long long`**, **`unsigned long long`**, **`float`**, **`double`**, **`long double`**.

- Если два балансируемых типа — **unsigned int** и **long** и тип **long** может представлять все значения типа **unsigned int**, результатом балансировки является тип **long**.
- В противном случае результатом балансировки является тип **unsigned long**.

Каждый из операндов преобразовывается в балансированный тип, арифметическая операция выполняется над значениями одинакового типа, и результат операции имеет балансированный тип.

1.2.3 Порядок вычисления выражения и побочные эффекты

Порядок, в котором транслятор вычисляет подвыражения неопределён и зависит от реализации. Например, оператор

```
y = *p++;
```

может быть эквивалентно либо

```
temp = p; p += 1; y = *temp;
```

либо

```
y = *p; p += 1;
```

Если в программе встретилось выражение

```
f() + g()
```

компилятор может расположить вызовы функций `f` и `g` в произвольном порядке.

При вызове функции, например `f(a, b)`, компилятор может вычислять выражения в произвольном порядке.

Порядок вычисления выражения важен, когда выражение имеет некоторый побочный эффект, например, заносит значение в переменную, либо модифицирует состояние файла.

Программа на Си содержит *точки согласования*. В точках согласования точно известно, какие побочные эффекты имели место, а какие должны произойти. Например, каждое выражение, записанное как оператор имеет точку согласования в конце. Гарантируется, что в фрагменте

```
y = 37;
x += y;
```

значение 37 будет помещено в `y` до того, как значение `y` будет использовано при вычислении `x`.

Точки согласования могут находится внутри выражения. Операции «запятая», вызов функции, логическое «и», логическое «или» содержат точки согласования. Например,

```
if ((c = getchar()) != EOF && isprint(c))
```

`isprint(c)` будет вычислено только после того, как новое значение, возвращённое `getchar()`, будет занесено в переменную `c`.

Между двумя точками согласования один и тот же объект может модифицироваться только один раз, и значение, читаемое из модифицируемого объекта может использоваться только для вычисления нового значения этого объекта.

Например,

```
val = 10 * val + (c - '0'); // хорошо
i   = ++i + 2;              // плохо
```

1.3 Многомерные массивы

Двухмерные массивы (матрицы) определяются следующим образом

```
<тип> <имя>[разм. 1][разм. 2];
```

Обратите, что каждое из измерений массива записывается в отдельной паре квадратных скобок. Массивы большей размерности определяются аналогично.

В памяти массивы размещаются по строкам, то есть быстрее всего изменяется последний индекс. Например для матрицы `int x[2][3]` элементы в памяти будут размещены следующим образом

```
x[0][0],x[0][1],x[0][2],x[1][0],x[1][1],x[1][2]
```

Поскольку матрица располагается по строкам, выражение `x[i]` имеет тип `int [3]`, то есть каждую строку матрицы можно рассматривать как целое (например, передавать параметром в функции). Аналогично и для массивов большей размерности.

Обращение к элементу массива выглядит так `x[i][j]`. Каждый индекс всегда записывается в своей паре квадратных скобок.

При передаче многомерных массивов в функции у них должны быть указаны все измерения, *кроме первого*. Это связано с тем, что для вычисления адреса элемента массива компилятор должен знать все измерения, кроме первого (напишите формулу вычисления адреса и проверьте!). Поэтому, функция матричного умножения может иметь следующий прототип.

```
double a[K][L], b[L][M], c[K][M];
void mult(double a[][L], double b[][M], double c[][M]);
```