1.14. Расширенные возможности языка

Friday, August 28, 2020 17:07

epc-1314-fa
sselt-c-ke...

В данном уроке мы рассмотрим расширенные возможности языка:

- Уменьшение использования оперативной памяти при помощи объединений (*unions*)
- Реализацию выражений через оператор-запятую (*comma operator*)
- Квалификаторы типа (*type qualifiers*): `register`, `volatile` и `restrict`
- Добавление поддержки аргументов командной строки
- Динамическое выделение оперативной памяти через `malloc()` и `calloc()` и ее освобождение через `free()`

Объединения (*unions*)

Объединение – довольно необычная конструкция. Она используется в тех случаях, когда нужно хранить разные типы данных в одном и том же участе памяти (читай: в одной переменной). Обычно так делают для экономии оперативной памяти.
Например, если нужно задать переменную `x`, которая будет использоваться для хранения символа, числа с плавающей точкой и целого числа, нужно сначала задать объединение. Назовем его `mixed`:

```
union  mixed
{
    char   c;
    float  f;
    int    i;
};
```

И затем объявим переменную-объединение:

```
union mixed  x;
```

Объявление объединения похоже на объявление структуры, просто используется другое ключевое слово: `union` вместо `struct`. Но главная разница между структурой и объединением – в организации памяти. Если бы это была структура, она содержала бы **три разных элемента**: `c`, `f`, `i`. Объединение же содержит только **один** элемент, который может быть `c`, `f`, **или** `i`.
Запись в объединение производится по аналогии со структурой. Запишем **символ** в `x`:

```
x.c = 'K';
```

Записанный символ можно прочитать так же по аналогии со структурой:

```
printf ("Character = %c\n", x.c);
```

Чтобы записать в x число с плавающей точкой, нужно написать `x.f`:

```
x.f = 786.3869;
```

Наконец, можно записать результат целочисленного деления переменной `count` на 2:

```
x.i = count / 2;
```

Так как все три элемента переменной `x` хранятся в одном и том же участке памяти, только **одно** значение может быть записано в неё. Также нужно следить за тем, чтоб извлекать данные в правильном формате, т.е. если записать в переменную `float` и прочитать эту переменную как `char`, ничем хорошим это не кончится.

Элементы объединений подчиняются тем же правилам арифметики, что и обычные переменные. То есть, выражение

```
x.i / 2
```

будет рассчитано по правилам целочисленного деления, потому что обе переменные целочисленные.

Объединение может содержать сколько угодно элементов. Занимаемое количество оперативной памяти будет равняться размеру **наибольшего** элемента. Объединения могут выступать в качестве элементов структур и массивов. При объявлении объединения не обязательно задавать ему имя, а также можно сразу после объявления задать переменные с этим объединением, по аналогии со структурами. Работа с **указателями** на объединения также ведется по аналогии со структурами.

Переменные-объединения можно инициализировать. Если элемент объединения не был явно указан, значение будет присвоего первому элементу объединения. Так, команда

```
union mixed  x = { '#' };
```

присвоит символ `'#'` первому элементу `x`, то есть `c`.

Также можно явно задать элемент, который нужно инициализировать. Команда

```
union mixed x = { .f = 123.456 };
```

задает элементу `f` объединения `x` значение `123.456`.

Можно присвоить переменной-объединению значение другой переменной такого же типа при инициализации:

```
void foo (union mixed x)
{
    union mixed  y = x;
    ...
}
```

Здесь функция `foo` присваивает переменной `y` значение аргумента `x`.

Объединения позволяют задавать массивы, **с элементами разного типа**. Например, конструкция

```
struct
{
    char              *name;
    enum symbolType   type;
    union
    {
        int     i;
        float   f;
        char    c;
    }                 data;
}  table [kTableEntries];
```

задает массив `table`, содержащий `kTableEntries` элементов. Каждый элемент массива содержит структуру, состоящую из указателя на строку `name`, перечисляемый тип `type` и объединение `data`, которое может содержать `int`, `float` или `char`. Элемент `type` можно использовать для хранения типа данных, используемого в объединении в составе каждого элемента массива. Чтобы записать символ `'#'` в `table[5]` и задать в поле `type`, что это символ, нужно написать две инструкции:

```
table[5].data.c = '#';
table[5].type = CHARACTER;
```

При переборе элементов `table`, можно легко определить тип данных, хранимый в каждом объединении. Например, программа ниже выведет все объединения массива table с учетом их типа:

```
1     enum symbolType { INTEGER, FLOATING, CHARACTER };
2
3        ...
4
5     for ( j = 0;  j < kTableEntries;  ++j ) {
6         printf ("%s  ", table[j].name);
7
8         switch ( table[j].type ) {
9             case INTEGER:
10                printf ("%i\n", table[j].data.i);
```

```
11                  break;
12              case FLOATING:
13                  printf ("%f\n", table[j].data.f);
14                  break;
15              case CHARACTER:
16                  printf ("%c\n", table[j].data.c);
17                  break;
18              default:
19                  printf ("Unknown type (%i), element %i\n", table[j].type, j );
20                  break;
21          }
22      }
```

Оператор-запятая (*comma operator*)

В уроке 1.3 (*Циклы*) мы в цикле `for` разделяли запятой условия цикла:

```
for ( i = 0, j = 100;  i != 10;  ++i, j -= 10 )
    ...
```

Запятую можно использовать для разделения выражений не только в циклах, но где угодно в коде. Выражения рассчитываются слева направо. Так, в программе

```
while ( i < 100 )
    sum += data[i], ++i;
```

значение `data[i]` прибавляется к `sum`, а затем инкрементируется `i`. Так как все операторы в C возвращают значение, оператор-запятая возвращает значение самого правого выражения.

Приведем еще одно полезное применение. Программу

```
1   string s;
2   read_string(s);
3   while(s.len() > 5)
4   {
5      //do something
6      read_string(s);
7   }
```

при помощи оператора-запятой можно сократить до

```
1   string s;
2   while(read_string(s), s.len() > 5)
3   {
4      //do something
5   }
```

Обратите внимание, что запятая для разделения аргументов при вызове функций или названия переменных при объявлении одного и того же типа – отдельная синтаксическая единица и **не является** примером применения оператора-запятой.

Квалификаторы типа (*type qualifiers*)

Квалификатор `register`

Если программе или функции нужно часто обращаться к какой-то переменной, можно запросить у системы ускоренный доступ к ней. Обычно это означает, что переменная будет помещена в один из **регистров** процессора на время выполнения функции. Скорость доступа к регистрам в десятки раз быстрее, чем к адресам оперативной памяти. Ключевое слово пишется при объявлении переменной:

```
register int   index;
register char  *textPtr;
```

Обычно только основные типы данных можно записать в регистры, а также указатели. Обратите внимание, что адресный оператор нельзя применять к регистрам, потому что переменная **не будет находиться** в оперативной памяти. Помимо этого, переменные `register` ведут себя как обычные автоматические переменные.

Также нет никаких гарантий, что компилятор вообще поместит переменную в регистр, это нужно проверять на каждой отдельной системе и компиляторе.

Квалификатор `volatile`

Этот квалификатор фактически противоположен `const`. Он используется для того, чтобы предотвратить оптимизацию переменной компилятором. Допустим, у нас есть порт ввода-вывода, на который указывает переменная `outPort`. Если нужно записать два символа в этот порт, мы напишем

```
*outPort = 'O';
*outPort = 'N';
```

Компилятор увидит, что идут подряд два присваивания и уберет первое присваивание, т.к. посчитает, что оно ни на что не влияет, оставив только последнее присваивание. Чтобы это не происходило, нужно объявить `outPort` как `volatile`:

```
volatile char  *outPort;
```

Квалификатор `restrict`

Как и `register`, это оптимизационная подсказка для компилятора и он имеет полное право ее игнорировать. Квалификатор используется, чтобы указать компилятору, что указатель является **единственным** указателем на некоторую переменную, – что других указателей на нее **не будет**. Это позволит компилятору ускорить работу программы за счет опускания некоторых проверок.

Квалификатор пишется после звездочки и до названия указателя:

```
1 int * aPtr;
2 int * restrict aPtr;
```

Разберем работу квалификатора на примере [3]. Есть функция `update()`, которая на вход принимает три указателя на целочисленные значения: `*a, *b, *c`.

```
1 void update (int *a , int *b , int * c )
2 {
3     * a += * c;
4     * b += * c;
5 }
```

Эта функция прибавляет значение **`c`** к **`a`** и **`b`**. Выполнение этой функции займет 5 инструкций:

```
1 update:
2     movl (%rdx), %eax
3     addl  %eax, (%rdi)
4     movl (%rdx), %eax
5     addl  %eax, (%rsi)
6     ret
```

Так как мы не делали никаких оптимизаций, процессор перед каждой операцией сложения будет обращаться к значению, на которое указывает ***`c`** (строки 2 и 4). Теперь применим квалификатор `restrict` к этому указателю:

```
1 void update_restrict (int *a , int *b , int * restrict c )
2 {
3     * a += * c;
4     * b += * c;
5 }
```

Фактически мы указали компилятору, что кроме указателя **`*c`** нет **никаких других путей** обратиться к значению, на которое он указывает, то есть это значение **не будет меняться** никаким другим способом кроме как через этот указатель. Это позволяет компилятору опустить повторное обращение к переменной, потому что он точно знает, что ее значение останется **прежним** и можно просто использовать ранее записанное в регистр `%rdx` значение (строка 2):

```
1 update_restrict:
2     movl (%rdx), %eax
3     addl  %eax, (%rdi)
4     addl  %eax, (%rsi)
5     ret
```

Сравним производительность функций `update` и `update_restrict`. Если выполнить 100000000000 циклов первой функции, это займет 249 сек. Во втором случае выполнение идет 83 сек, то есть функция работает в 3 раза быстрее. Скорость достигается не только за счет меньшего количества инструкций, но и за счет уменьшения количества обращений к оперативной памяти (вспомните квалификатор `register`).

Помните, что если квалификатор `restrict` был применен на указатель, но при этом **существуют** другие пути изменить переменную, на которую он указывает, то это может привести к **ошибкам**. В моем случае компилятору хватило ума увидеть ошибку, исправить ее и выдать предупреждение:

```
warning: passing argument 3 to restrict-qualified parameter aliases with argument 1
[-Wrestrict]
```

Но давайте разберем, что произойдет, если компилятор не исправит ошибку. Напишем программу:

```
1  int main ( void )
2  {
3      int a = 1;
4      int b = 2;
5
6      update (&a, &b, &a) ;
7      printf ("Expected Result: %d %d\n", a, b) ;
8
9      a = 1; b = 2; // reset values
10
11     update_restrict (&a, &b, &a) ;
12     printf ("Actual Result: %d %d\n", a, b) ;
13 }
```

Функция update сначала рассчитывает `*a = *a + *a = 1 + 1`. Теперь `*a = 2`. Затем рассчитывается `*b = *a + *b = 2 + 2`. Теперь `*b = 4`. Полученные значения 2 и 4 выводятся на экран.

Функция `update_restrict` ведет себя немного иначе. Как уже сказали выше, функция теперь не перепроверяет значение третьего аргумента **`*c`** при расчете второго выражения. Но значение **`*a`**, которое подставляется на место **`*c`**, будет изменено в результате выполнения первого выражения. А второе выражение об этом не узнает. И рассчитает **`*b`**, исходя из того, что `*a = 1`, потому что именно это значение было изначально записано в регистр. Таким образом, функция выдаст 2 и 3 вместо 2 и 4.

Ключевое слово **`inline`**

Это слово по функционалу схоже с макросами. В большинстве языков программирования принято разбивать программу на отдельные функции. Хоть это и упрощает работу с кодом и уменьшает размер программы, у функций есть один недостаток: процессору нужно тратить время на переход к вызываемым функциям, – нужно выделить место в стеке, сохранить адрес возврата и передать параметры в стек.

Чтобы решить эту проблему, можно вместо переходов к одной функции продублировать код функции в те места, где одна нужна. Так процессору не нужно будет переходить к функции, а он просто будет выполнять функцию последовательно с основным кодом. Для этого и используется

inline, – чтобы показать компилятору, что оптимальнее будет выполнить подстановку кода функции туда, где одна вызывается.

Помните о том, что лучше так делать **только с маленькими функциями**, потому что для подобной работы с большими функциями процессору **может не хватить регистров**, чтобы хранить все переменные одновременно.

Приведем пример применения слова `inline`. Напишем `inline`-функцию и вторую функцию, которая будет ее вызывать:

```
1 inline static int add (int s1, int s2)
2 {
3     return s1 + s2;
4 }
5
6 int some_function (int a, int b)
7 {
8     return add (a, b);
9 }
```

Такой код после компиляции трансформируется в

```
1 int some_function (int a, int b)
2 {
3     return a + b;
4 }
```

Сравним прирост производительности, который даст использование `inline`. Без этого слова выполнение 100000000000 итераций функции `some_function()` занимает 67.5 сек. Если функцию `add()` переделать в `inline`, время выполнения сокращается до 22.3 сек, что в 3 раза быстрее.

У `inline`-функций есть некоторые **ограничения**. Например, они **не могут быть рекурсивными**, потому что при компиляции невозможно определить, сколько раз будет вызвана функция (т.е. сколько раз подряд ее нужно подставить).

Слово `inline` является лишь **намеком** компилятору, но никак не директивой и потому он может его проигнорировать. Если нужно точно убедиться, что компилятор не проигнорирует `inline` и создаст `inline`-функцию, нужно использовать дополнительные **атрибуты**. В случае компилятора **GCC** это атрибут `__attribute__(( always_inline))`. Есть и противоположный ему атрибут, – `__attribute__((noinline))`, который гарантирует то, что функция не будет `inline`. Применение атрибутов выглядит так:

```
inline __attribute__ (( always_inline )) int inline_function (int a , int b ) ;

inline __attribute__ (( noinline ) ) int no_inline_function ( int a , int b ) ;
```

Ключевое слово **extern**

Это ключевое слово указывает на то, что переменная **уже была определена ранее** в других файлах программы и благодаря этому компилятор не будет несколько раз выделять память под одну и ту же переменную. До этого момента мы не разделяли понятие "объявление" и "декларирование" переменной. Чтобы задекларировать переменную, мы писали

```
int i;
```

что не только декларирует переменную `i`, но и объявляет ее, т.е. компилятор выделяет под нее место в памяти. Чтобы задекларировать переменную без ее объявления, нужно написать

```
extern int i;
```

что проинформирует компилятор, что такая переменная уже была задана ранее и под нее не нужно

выделять память.

Слово `extern` работает со всеми типами данных. Когда оно применяется к массиву, не обязательно указывать размеры массива, потому что размеры уже были указаны ранее при декларировании:

```
extern int a[];
```

Обычно `extern` используется для доступа к одной и той же переменной из разных файлов программы, поэтому ее рекомендуется помещать в **заголовок** программы, чтобы их сразу было видно.

Динамическое выделение оперативной памяти

Когда мы объявляем переменную, будь это простым типом данных, массивом или структурой, под хранение значения этой переменной компилятор всегда выделяет участок оперативной памяти.

Часто необходимо **динамически** выделять память по ходу работы программы. Допустим, наша программа читает данные из файла в массив. И мы не знаем, сколько данных записано в файле. Есть три варианта реализации такой программы:

- Задать массив настолько большой, насколько это позволяет компилятор
- Использовать массив переменной длины и задать ему длину при выполнении программы
- Динамически выделить место под массив

Первый вариант даже комментировать не стоит. Это наименее оптимальный способ.

Второй вариант получше, но в случае с массивами переменной длины нет никаких способов освободить занимаемую ими память.

Третий вариант обычно наиболее оптимальный, так как позволяет выделять и очищать память. Его и разберем подробнее.

Функции `calloc()` и `malloc()`

Эти функции используются для выделения памяти при работе программы и находятся в составе стандартной библиотеки C, – `<stdlib.h>`. Поэтому её нужно включать в программу перед их использованием.

Функция `calloc()` принимает два аргумента: первый указывает требуемое количество элементов, второй – размер каждого элемента в **байтах**. Затем она **обнуляет** выделенное пространство и возвращает указатель типа `void` на **начало** выделенного пространства. Перед тем как записать значение указателя, можно задать ему правильный тип через type cast оператор.

Функция `malloc()` работает схожим образом, но принимает только один аргумент, – суммарное количество байт, которое нужно выделить. Также он **не обнуляет** выделенную область в отличие от `calloc()`.

При недостатке памяти обе функции вернут `NULL` (**нулевой указатель**).

Оператор `sizeof`

Этот оператор можно использовать для определения объема памяти (в байтах), выделенного под объект. Это может быть название базового или производного типа данных, выражением, переменной, массивом, областью, выделенной через `malloc`/`calloc` и т.д. Например,

```
sizeof (int)
```

выдаст количество байт, выделяемое под целочисленную переменную, то есть 4. Если объявить массив `x` из 100 таких элементов, то

```
sizeof (x)
```

выдаст 400, соответственно. Выражение

```
sizeof (struct dataEntry)
```

покажет количество памяти, требуемое для хранения одного элемента структуры `dataEntry`. Ну а если объявить массив `data` с такими элементами, то выражение

```
sizeof (data) / sizeof (struct dataEntry)
```

покажет количество элементов в этом массиве. Выражение

```
sizeof (data) / sizeof (data[0])
```

выдаст тот же результат. Такую конструкцию для удобства можно упаковать в макрос:

```
#define  ELEMENTS(x)   (sizeof(x) / sizeof(x[0]))
```

и затем легко получать количество элементов в массиве:

```
if ( i >= ELEMENTS (data) )
    ...
```

или

```
for ( i = 0; i < ELEMENTS (data); ++i )
    ...
```

Кстати, это не работает внутри функций. То есть, вот так нельзя:

```
1 int f (int a [] )
2 {
3      int len = sizeof(a) / sizeof (a [0] ) ;
4      /*** WRONG: not the number of elements in a ***/
5 }
```

потому что функция получает на вход не сам массив, а **указатель на массив** и оператор `sizeof` выдаст размер этого указателя. То есть, в случае 64-битной системы размер указателя будет 64 бита (8 байт), а `int` занимает 4 байта, т.е. функция всегда будет выдавать 8 / 4 = 2.

Помните, что `sizeof` – это оператор, а не функция, несмотря на то, что он похож на функцию. Его значение рассчитывается **при компиляции**, а не во время работы программы, за исключением случая, когда в качестве аргумента передается **массив переменной длины**.

Вернемся к динамическому выделению памяти. Допустим, нужно выделить память под хранение 1000 `int`-ов:

```
#include <stdlib.h>
  ...
int  *intPtr;
 ...
intPtr = (int *) calloc (sizeof (int), 1000);
```

Через `malloc()` будет так:

```
intPtr = (int *) malloc (1000 * sizeof (int));
```

Помните, что как `malloc()`, так и `calloc()` возвращают указатель типа **void** и ему нужно задавать **правильный тип** через type cast, что и делается в примерах выше, – задается тип "указатель на `int`".

Как уже говорили, обе функции вернут `NULL` (**нулевой указатель**), если запрошенного количества памяти будет недостаточно. Рекомендуется **всегда** проверять этот момент. Код ниже выделяет место под 1000 указателей на `int` и проверяет успешность операции.

```
1 #include <stdlib.h>
2 #include <stdio.h>
3    ...
4 int  *intPtr;
5    ...
```

```
intPtr = (int *) calloc (sizeof (int), 1000);

if ( intPtr == NULL )
{
    fprintf (stderr, "calloc failed\n");
    exit (EXIT_FAILURE);
}
```

Если операция успешна, переменную `intPtr` можно использовать так, будто она указывает на массив из 1000 int-ов. То есть, чтобы задать всем элементам значение –1, можно написать:

```
    for ( p = intPtr; p < intPtr + 1000; ++p )
        *p = -1;
```

где `p` – указатель на `int`.

Чтобы выделить место под n элементов типа `struct dataEntry`, сначала нужно задать указатель соответствующего типа:

```
struct dataEntry  *dataPtr;
```

и затем вызвать функцию calloc() для выделения памяти:

```
dataPtr = (struct dataEntry *) calloc (n, sizeof (struct dataEntry));
```

Ну и не забывать про проверку, что возвращенный указатель **не является нулевым**. Теперь указатель может быть использован в программе, как будто он указывает на **массив** из n элементов `dataEntry`. Например, если в структуре `dataEntry` есть элемент `index`, можно задать ему значение 100:

```
dataPtr->index = 100;
```

Функция **free**

Когда мы закончили работать с памятью, которая была выделена через `malloc()` или `calloc()`, ее можно освободить при помощи функции `free()`. Единственный нужный ей аргумент – указатель на **начало** выделенной области, – который был возвращен функцией `calloc()` или `malloc()` после её выделения. Если продолжать пример выше, использование функции будет выглядеть так:

```
free (dataPtr);
```

Функция `free()` не возвращает никаких значений.

Динамическое выделение памяти – очень ценный инструмент при работе со связными структурами, такие как **связные списки**. Когда нужно добавить новый элемент в список, можно **динамически** выделять память под новый элемент и с легкостью создавать ссылку на этот элемент, потому что функции `calloc()` и `malloc()` как раз возвращают указатель на него. Допустим, что указатель `listEnd` указывает на конец списка типа `struct entry`:

```
struct entry
{
    int             value;
    struct entry    *next;
};
```

Ниже функция `addEntry()` получает в качестве аргумента указатель на начало связного списка и добавляет новый элемент в список.

```
#include <stdlib.h>
#include <stddef.h>

// add new entry to end of linked list

struct entry *addEntry (struct entry *listPtr)
{
    // find the end of the list
```

```
 9
10          while ( listPtr->next != NULL )
11              listPtr = listPtr->next;
12
13          // get storage for new entry
14
15          listPtr->next = (struct entry *) malloc (sizeof (struct entry));
16
17          // add null to the new end of the list
18
19          if ( listPtr->next != NULL )
20               (listPtr->next)->next = (struct entry *) NULL;
21
22          return listPtr->next;
23      }
```

Если добавление успешно, функция добавляет нулевой указатель в поле `next` нового элемента и возвращает указатель на этот элемент. Если выделить память не получилось, будет возвращен нулевой указатель.

Функция **realloc()**

Бывает, что нужно высвободить или добавить **часть** памяти к уже существующему участку. Функция `realloc` как раз для этого предназначена. Она принимает на вход два параметра: указатель на изменяемый участок и новый размер участка и возвращает указатель на новое начало участка. Данные при этом не обнуляются, по аналогии с `malloc`.

```
void *realloc(void *ptr, size_t size);
```

Помните, что в `realloc` нужно передавать указатель, выданный предыдущим вызовом `malloc`, `calloc` или `realloc`, иначе результат будет непредсказуемым. Новый размер участка памяти может быть как больше, так и меньше исходного. Также после выполнения функция выдает новый указатель на участок и поэтому нужно не забыть его **обновить** везде, где он используется.

Стандарт C приводит несколько правил по поводу работы `realloc`:

- При расширении участка памяти, realloc не инициализирует добавляемые байты
- Если `realloc` не может выделить запрошенное количество памяти, он возвращает нулевой указатель. При этом расширяемый участок остается без изменений.
- Если в `realloc` в качестве первого аргумента передать нулевой указатель, он будет вести себя как malloc.
- Если в `realloc` в качестве второго аргумента передать 0, он будет вести себя как `free`.

К сожалению, как именно работает `realloc`, не указывается. Но в целом, когда размер участка уменьшается, функция обычно обрезает существующий участок, чтобы сохранить оставшиеся данные. Точно также она пытается увеличить участок, не перемещая его. Если же так сделать не получается, она выделит участок в другом месте и скопирует данные из старого участка в новый.

Проблема "висячих указателей"

Операции очистки памяти приводят к новой проблеме, – **висячие указатели** (*dangling pointers*). Вызов функции `free(p)` не очищает участок памяти, а просто высвобождает его. И нужно помнить о том, что все указатели, которые указывали на этот участок памяти, уже **не будут валидными**, там с высокой долей вероятности будет **мусор**, потому что освобожденный участок может быть спустя какое-то время перезаписан другими данными. То есть, вот так нельзя:

```
char *p = malloc(4);
     ...
free(p);
     ...
```

```
    strcpy(p, "abc");  /*** WRONG ***/
```

Проблема висячих указателей в том, что на один и тот же участок памяти может указывать несколько указателей и за всеми ними нужно следить, потому что после освобождения участка **они все станут висячими**.

Аргументы командной строки и операции с файлами

ПРОПУЩЕНО

Источники:

1. Stephen Kochan – Programming in C (4th Edition); chapter 16
2. What does the comma operator , do? – https://stackoverflow.com/questions/52550/what-does-the-comma-operator-do
3. restrict, static & inline Keywords in C – Markus Fasselt (см. приложенный к уроку документ)
4. Why is volatile needed in C? – https://stackoverflow.com/questions/246127/why-is-volatile-needed-in-c