Постановка задачи

Формулировка задания

1. На языке C разработать функцию, реализующую определенную вариантом задания функциональность. Поместить определение функции в отдельный исходный файл, оформить заголовочный файл. Разработать тестовую программу на языке C.
2. Собрать программу «по шагам». Проанализировать выход препроцессора и компилятора. Проанализировать состав и содержимое секций, таблицы символов, таблицы перемещений и отладочную информацию, содержащуюся в объектных файлах и исполнимом файле.
3. Выделить разработанную функцию в статическую библиотеку. Разработать make-файлы для сборки библиотеки и использующей ее тестовой программы. Проанализировать ход сборки библиотеки и программы, созданные файлы зависимостей.

Формулировка варианта задания

Перестановка местами элементов массива с четными и нечетными индексами (в случае нечетного числа элементов, последний элемент игнорируется).

Пример

Пусть элементы массива типа **unsigned** имеют следующие значения (в порядке возрастания индекса, начиная с 0): 0, 1, 2, 3, 4, 5, 6. После выполнения программы в том же массиве должны располагаться числа: 1, 0, 3, 2, 5, 4, 6 (переставлены 0-й и 1-й элементы, 2-й и 3-й, 4-й и 5-й).

Решение задачи

Шаг 1. Прототип разрабатываемой функции

Как легко догадаться, для выполнения требуемых действий разрабатываемая «*функция*» (**function**) – термин языка C для подпрограммы - должна получать информацию о количестве элементов массива и о местонахождении их в памяти.

«Местонахождение» элементов массива естественно определять адресом его первого элемента. В языке C адреса объектов в памяти представляются *указателями* (**pointer**); *значением* (**value**) указателя является адрес некоторого объекта в памяти, *тип* (**type**) этого объекта известен компилятору благодаря тому, что ему известен тип указателя, и «знание» типа объекта позволяет компилятору формировать подходящие последовательности инструкций для реализации *доступа* (**access**) к объекту.

Количество элементов массива – это просто целое число без знака подходящей (достаточно большой) разрядности. Разрабатываемая функция предназначена для обработки массива, *отображенного* (**mapped**) в адресное пространство процесса («исполняемого экземпляра» программы)[1]. Очевидно, количество элементов массива – его *длина* (**length**) не может превышать

[1] В простейшем случае это означает, что массив целиком находится в оперативной памяти (main memory) машины, однако при наличии механизма виртуальной памяти (virtual memory) массив может иметь размер больший, чем весь доступный объем оперативной памяти, и в этом случае в каждый момент времени в оперативной памяти будет находиться только его часть (или вовсе ничего), остальные же данные будут находиться во «вторичной» памяти (secondary memory) – обычно, на диске.

его *размера* (**size**) – числа *адресуемых* (**addressable**) ячеек памяти, содержащих данные массива; в то же время, в некоторых архитектурах эти числа могут совпадать (если каждый элемент массива содержится в точности в одной адресуемой ячейке памяти). Стандарт языка C предусматривает определение типа `size_t` – типа беззнаковых чисел разрядности, достаточной для представления размера любого объекта, учитывая сказанное выше, именно этот тип естественно использовать для представления количества элементов массива. Следует отметить, что тип `size_t` не является *базовым* (**basic**) типом языка, а определяется в *стандартном заголовочном файле* (**standard header**) `<stddef.h>`, следовательно, этот заголовочный файл должен быть *включен* (**include**) в исходный текст перед использованием типа `size_t`.

В языке C для передачи информации в вызываемую функцию естественно использовать параметры[2], таким образом, разрабатываемая функция должна иметь 2 параметра: указатель на первый элемент массива (соответствующего типа) и его длину (типа `size_t`). Результатом работы функции является *побочный эффект* (**side effect**) – изменение содержимого памяти, функция не имеет естественного *возвращаемого значения* (**return value**).

Теперь, учитывая все сказанное, мы можем записать *прототип* разрабатываемой *функции* (**function prototype**) – *объявление* (**declaration**), содержащее типы параметров функции и тип ее результата:

```
#include <stddef.h>

void swap_pairs( /*inout*/ unsigned *array, size_t array_length );
```

Шаг 2. Тело функции

Реализация требуемой функциональности обеспечивается следующим определением:

```
#include <stddef.h>

void swap_pairs( /*inout*/ unsigned *array, size_t array_length ) {
    const size_t pair_count = array_length / 2;
    void * const end_ptr = array + pair_count * 2;
    for( unsigned *array_ptr = array;
         array_ptr != end_ptr;
         array_ptr += 2 ) {
        const unsigned t = * array_ptr;
        * array_ptr = * ( array_ptr + 1 );
        * ( array_ptr + 1 ) = t;
    }
}
```

Прежде всего, рассчитывается количество обрабатываемых пар элементов, при этом используется следующее свойство операции целочисленного деления, закрепленное действующим стандартом язык: результатом деления целых чисел является частное с *отброшенной* (**discarded**) дробной частью, отбрасывание дробной части также называют *усечением в сторону нуля* (**truncation toward zero**). Полученное количество пар размещается в локальной переменной `pair_count`.

[2] Возможны и другие способы передачи информации в функцию, например, через глобальные переменные.

Обработка массива выполняется в цикле, при этом для доступа к элементам используется указатель `array_ptr`. В каждой итерации цикла значением указателя является адрес первого элемента очередной пары смежных элементов: в первой итерации значением `array_ptr` будет адрес 0-го элемента массива, во второй – 2-го элемента и т.д. Для этого указатель `array_ptr` инициализируется значением указателя `array`, как раз равным адресу 0-го элемента массива. *В конце* каждой итерации указатель `array_ptr` должен модифицироваться таким образом, чтобы его значением был адрес первого элемента следующей пары. Требуемая величина изменения адреса определяется при этом размером элемента массива, выраженным в числе адресуемых ячеек памяти. Например, если массив составлен из машинных слов и в машине используется *словная адресация* (**word-addressable machine**), значение адреса должно каждый увеличиваться на два. Напротив, если используется байтовая адресация (**byte-addressable machine**), и размер машинного слова составляет, например, 4 байта, значение адреса должно увеличиваться на 8 (2·4). Язык C, будучи языком высокого уровня, позволяет программисту отвлечься от этих деталей и разрабатывать код, который с одинаковым успехом может исполняться на самых разных вычислительных машинах. В частности, операция прибавления 1 к указателю определена таким образом, что значением указателя становится адрес следующего элемента массива, из этого уже само собой вытекает определение операций прибавления и вычитания из указателя произвольного целого числа. Таким образом, в нашем случае требуемая модификация *указателя* обеспечивается прибавлением к нему числа 2, при компиляции кода данной операции будут поставлены в соответствие инструкции прибавления к *адресу* числа 2, 8 или иного, в зависимости от *целевой машины* (**target**).

Цикл обработки массива должен выполняться до тех пор, пока не будет обработано требуемое количество пар элементов. При обработке `pair_count` пар операция прибавления числа 2 к указателю `array_ptr` будет выполнена `pair_count` раз; следовательно, результирующее значение `array_ptr` можно вычислить заранее, и *отличие* значения `array_ptr` от этого результирующего значения может служить признаком необходимости *продолжит*ь выполнение цикла. В приведенном определении результирующее значение `array_ptr` помещается в локальную переменную-указатель `end_ptr`. Очевидно, указатель `end_ptr` не должен использоваться для доступа к памяти, поэтому мы используем для него тип `void` («пусто»).

В теле цикла выполняется перестановка пары смежных элементов массива. Значением указателя `array_ptr` является адрес первого элемента пары, следовательно, адрес второго элемента является значением указателя `(array_ptr + 1)`. Для чтения и записи элементов массива используется операция *разыменовывания* (**dereference**) указателя – *унарная* (**unary**) – то есть с одним операндом - операция “*”.

В приведенном коде несколько раз встречается *квалификатор* (**qualifier**) `const`, цель использования которого состоит в обозначении объектов с неизменными значениями. Так, значение `pair_count`, будучи единожды рассчитанным, не должно изменяться. Аналогично, не должно изменяться значение указателя `end_ptr`[3]. В теле цикла переменная `t` используется для хранения значения первого элемента обрабатываемой пары, ее значение также не должно изменяться; то, что при этом в каждой итерации переменная `t` получает новое значение не должно сбивать с толку: время жизни (**lifetime**) этой переменной длится от момента ее объявления до завершения блока, то есть в каждой итерации «рождается» новая переменная `t`.

[3] Следует обратить внимание на положение квалификатора `const` в этом случае – определяется «константный указатель», а не указатель на константу.

Шаг 3. Тестовая программа

Тестовая программа должна определять тестовый массив и вызывать функцию `swap_pairs` для его обработки.

Для начала, определим тестовый массив:

```
unsigned test_array[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
```

Отметим, что в приведенном определении длина массива не указывается явно (в квадратных скобках), а подсчитывается компилятором по результатам анализа *списка инициализации* (**initializer list**).

В программах, разработанных на языке C, после инициализации *среды исполнения* (**runtime**) управление передается функции `main`, которую в рассматриваемом случае можно определить следующим образом[4]:

```
int main( void ) {
    const size_t array_length =
        sizeof( test_array ) / sizeof( test_array[ 0 ] );
    swap_pairs( test_array, array_length );
    return 0;
}
```

В приведенном определении `main` не имеет параметров и возвращает целое число, которое служит *кодом завершения* (**termination status**) процесса.

При вызове функции `swap_pairs` необходимо передать ей 2 аргумента: указатель на 0-й элемент обрабатываемого массива и его длину.

Длина тестового массива, разумеется, известна компилятору, но в языке C с этим числом не связан какой-либо идентификатор; в результате, длина массива должна или *зашиваться* (**hardcode**) в текст программы, или вычисляться. В данном случае для передачи функции `swap_pairs` длины обрабатываемого массива мы могли бы просто указать число 11, однако при этом определение массива `test_array` и использованный *литерал* (**literal**) – последовательность символов, представляющая целочисленную константу – оказались бы *неявно* (**implicitly**) связанными: при изменении определения массива следовало бы соответствующим образом изменить литерал. Неявный характер этой связи означает, что она неизвестна компилятору, следовательно, он не может контролировать требуемое соответствие. Таким образом, лучшим решением является вычисление длины массива, примененное в приведенном коде. Очевидно, длина массива может быть вычислена как отношение размера массива к размеру (любого) его элемента, а требуемые размеры могут быть определены с помощью оператора `sizeof`. Заметим, что размеры массива и его элемента известны компилятору, поэтому обычно длина массива будет вычислена на этапе компиляции, а не исполнения, таким образом, «вычисление» длины массива на самом деле не влечет накладных расходов процессорного времени.

Указатель на 0-й элемент массива `test_array` является результатом выражения `&test_array[0]` (здесь “&” – унарная операция взятия адреса), однако в языке C существует

[4] В программах, принимающих аргументы командной строки, функция main определяется иначе: функция принимает два параметра – количество переданных аргументов командной строки и массив указателей на строки, содержащие эти аргументы.

тесная связь между массивами и указателями, и почти во всех контекстах значение идентификатора массива *конвертируется* в указатель на его 0-й элемент, в результате допустима более короткая запись, которая обычно и используется в программах: `test_array` вместо `&test_array[0]`. Следует обратить внимание, что, например, в выражении `sizeof( test_array )` идентификатор `test_array` в `&test_array[0]` не преобразуется, иначе результатом выражения был размер указателя[5], а не размер массива; к сожалению, подобные особенности языка C осложняют его освоение и в некоторых случаях затрудняют понимание исходного текста на этом языке.

Save

Поскольку содержимое массива нам придется распечатать 2 раза (до и после обработки), разумно создать для этого функцию. При этом имеется 2 возможности: функция может распечатывать содержимое конкретного массива `array` или произвольного массива, переданного ей в качестве параметра. Мы выберем второй вариант, который, прежде всего, приводит к меньшей «связности» (зафиксированной в исходном тексте взаимной зависимости) кода, а также открывает возможности повторного использования разработанной функции:

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

void print_uint_array( const unsigned *array, size_t array_length ) {
    for( size_t i = 0; i < array_length; ++ i ) {
        printf( "[ %zu ] %u\n", i, array[ i ] );
    }
}
```

Естественными параметрами функции вывода массива являются указатель на его 0-й элемент и длина. Для вывода массива необходимо иметь возможность чтения, но не записи его элементов; учитывая это, разумно использовать в качестве параметра функции указатель на *константные* данные: во-первых, это предотвратит возможность перезаписи данных массива в случае ошибочной реализации функции, во-вторых, это позволит использовать функцию для вывода не только изменяемых массивов, но и константных. Таким образом, параметр `array` имеет тип «указатель на константу типа `unsigned`» (обратите внимание на положение квалификатора `const` в объявлении параметра).

Для вывода содержимого массива организуется цикл перебора индексов его элементов, в теле которого выполняется печать индекса и значения элемента. Мы используем функцию *форматного вывода* (**formatted output**) `printf`, реализация которой предоставляется стандартной библиотекой языка. Объявление этой функции содержится в стандартном заголовочном файле “stdio.h”, который необходимо *включить* (**include**) в исходный текст *перед* использованием идентификатора `printf`, так что, встретив этот идентификатор, компилятор

[5] В случае RISC-V – 4 или 8 байт.

будет «знать», что это имя функции с определенным набором параметров и типом возвращаемого значения.

Стандартная функция `printf` является функцией с *переменным списком параметров* (**variable argument list**). Первый параметр функции является строкой (технически, в языке С строка является массивом символов, завершающимся символом с кодом 0, так что первым параметром `printf` является не строка, а указатель на 0-й элемент такого массива), которая определяет, что будет печататься в ходе выполнения функции. Строка может содержать непосредственно выводимые символы (например, первые два символа "[" (квадратная скобка и пробел) будут выведены в стандартный поток вывода) и *спецификаторы преобразований* (**conversion specification**), каждый из которых приводит к печати очередного аргумента функции `printf`. Спецификатор преобразования является последовательностью 2 и более символов, начинающейся с символа "%", последовательность определяет тип соответствующего аргумента, а также формат его вывода; в связи с этим, первый аргумент функции `printf` называется *форматной строкой* (**format string**). В рассматриваемом примере имеется 2 спецификатора преобразования: "%zu" и "%u". *Первый* спецификатор означает, что *первый* после форматной строки аргумент (т.е. *второй* аргумент функции) имеет тип `size_t`[6] и должен быть напечатан в виде десятичного целого. *Второй* спецификатор означает, что *второй* после форматной строки аргумент (*очередной* – третий - аргумент функции) имеет тип `unsigned` и также должен быть напечатан в виде десятичного целого.

Пара символов "\n" образует *управляющую последовательность* (**escape sequence**), *представляющую* (**represent**) *один* специальный символ – символ *перевода строки* (**line feed**). Например, в кодировках ASCII и Unicode эта последовательность используется для записи символа с кодом 10 (0xA). Преобразование управляющей последовательности в код символа осуществляется на этапе трансляции, то есть на этапе выполнения массив, указатель на 0-й символ которого передается в качестве 1-го аргумента функции `printf`, будет состоять из 12 символов, причем предпоследним символом массива будет символ перевода строки, а последний символ будет иметь код 0 (см. выше).

Отметим, что функция `printf` возвращает значение типа `int` (целое число со знаком) – ее результатом является количество напечатанных символов или, в случае ошибки, отрицательное число. В рассматриваемом примере значение, возвращаемое функцией `printf`, не используется (игнорируется).

В названии функции "uint" является сокращением от "unsigned int". Дело в том, что в языке С "unsigned" является просто сокращенной записью полного названия базового типа - "unsigned int". То есть, везде, где мы пишем `unsigned`, можно было бы писать `unsigned int`.

---

[6] Модификатор "z" появился в C99, но *до сих пор* поддерживается не всеми реализациями языка (в частности, реализацией стандартной библиотеки компании Microsoft).

Имеющееся определение функции мы можем поместить в отдельный файл. Имя файла может быть выбрано произвольно, однако *принято* использовать для файлов, содержащих определения функций и данных на языке C, постфикс “.c”. Поскольку в нашем определении используется тип `size_t`, не являющийся базовым, необходимо также дать компилятору определение этого типа, *включив* (**include**) в наш файл стандартный заголовочный файл “stddef.h”. В языке C, используемый тип должен быть предварительно определен (или объявлен), следовательно, файл “stddef.h” следует включить перед определением:

```
// swap_pairs.c

#include <stddef.h>


void swap_pairs( /*inout*/ unsigned *array, size_t array_length ) {
    const size_t pair_count = array_length / 2;
    void * const end_ptr = array + pair_count * 2;
    for( unsigned *array_ptr = array;
         array_ptr != end_ptr;
         array_ptr += 2 ) {
        const unsigned t = * array_ptr;
        * array_ptr = * ( array_ptr + 1 );
        * ( array_ptr + 1 ) = t;
    }
}
```