Типы данных в C

Мы переходим к более глубокому изучению языка чтобы писать на нём правильно и профессионально. В этом модуле мы сконцентрируемся на системе типов в C. Опытные программисты используют типы в C совершенно иначе, нежели новички. Пока что мы видели всего несколько типов данных:

- Два численных типа:
  - `int` для целых чисел (его размер отличается на разных платформах)
  - `char` для целых чисел размера один байт; строчки кодируются как массивы чисел типа `char`, каждое из которых хранит код символа по таблице ASCII.
- Типы-указатели. От любого типа T можно образовать тип-указатель T*. Данные типа T* хранят адреса данных типа T. Это правило можно применять сколько угодно раз, например, `int*` хранит адрес `int`, `int**` хранит адрес `int*` и т.д.
- Массивы, множество элементов лежащих в памяти один за другим. Массивы во многом похожи на указатели: имя массива можно использовать как указатель на его первый элемент.

Мы уже заметили, что типы данных указываются явно в том месте, где данные объявляются. Также мы знаем, что во время выполнения программы никакой информации о типах не сохраняется: мы не можем имея только адрес данных точно определить их тип.

Стандарт языка

С этого момента мы будем часто обращаться к стандарту языка. Существует линейка стандартов, каждый из которых является эволюцией предыдущих: C89 (также ANSI C), C99, C11, C18. Мы используем последнюю версию стандарта, C18; она отличается от C11 только исправлением ошибок и неточностей.

Стандарты постоянно совершенствуются, черновики новых версий стандарта распространяются бесплатно. Затем, когда стандарт фиксируется, его финальная версия распространяется за деньги; однако последний черновик стандарта практически ничем от него не отличается. На момент написания этого курса последним таким черновиком был ISO/IEC 9899:2017, на него мы и будем опираться.

Стандарт языка содержит часть его описания, общую для всех платформ, на которых язык может использоваться. Полезно уметь с ним работать, так как он помогает разобраться в функционировании даже таких программ, которые выглядят необычно.

Классификация типов

На деле типов в языке достаточно много. Их можно поделить на следующие категории:

- Встроенные численные типы: `int`, `unsigned int`, `unsigned shortint`, `char`, `float` и т.д.
- Массивы.
- Тип `void`, у которого только одно значение, с которым нельзя ничего сделать [1].

- Указатели (в том числе на функции – этого мы ещё не видели).
- Структурные типы – "пачки" из фиксированного количества элементов разных типов. В математике их называют *кортеж*.
- Объединения – типы данных, для которых указываются несколько других типов, и данные типа объединения могут быть одним из них.
- Перечисления – тип, для которого описывается фиксированное количество целочисленных значений, а данные принимают одно из этих значений.
- Типы функций.
- Псевдонимы для других типов.
- Типы, образованные от других типов с помощью модификаторов `const`, `restrict`, `volatile` ...

Этот устрашающий список не означает, что вам нужно пользоваться всеми типами. Многие, такие, как разные вариации типа `int` (`short int`, `long int`, `long long int` и т.д.) являются артефактом давно ушедшей эпохи. В силу того, как C появился (язык для написания операционной системы для слабого компьютера и игрушки в качестве хобби), как происходило продумывание его возможностей (не производилось), как он развивался (стихийно), такие типы используются в существующих программах, поэтому о них надо знать; в то же время в современных версиях языка им появились лучшие альтернативы.

Например, на разрядность и диапазон значений типа `short int` наложены нестрогие ограничения; в пределах этих ограничений и разрядность, и диапазон могут быть любыми. Иллюстрация этого факта послужит также первым примером работы со стандартом языка на одном из следующих слайдов.

Хороший код не использует устаревшие типы, оставшиеся в C от ушедших времён; в то же время, он активно использует составные типы и модификаторы `const`, `restrict` и т.д. чтобы программы было проще писать, читать и отлаживать. Мы об этом расскажем.

[1] В теории типов (это математическая база для типов в языках программирования) тип `void` из C обычно называют Unit. Unity в переводе с английского это единение, единица; возможное значение у типа `void` в C только одно, и никаких действий с ним производить нельзя.

Численные типы

Целые числа это центральный тип для языка C. Они бывают разного размера, со знаком (могут быть отрицательными) и без знака (только ноль или положительные). Сначала мы расскажем об их особенностях, потому что в чужом коде вы их будете видеть часто; затем мы покажем, как в современном C обходиться без большинства этих типов.

Тип `char`

- Может быть знаковым (`signed char`) или беззнаковым (`unsigned char`). Стандарт языка не описывает, что будет, если мы напишем просто `char`, для большинства компиляторов `char` это синоним `signed char`.

- Размер всегда равен одному байту.
- Хотя имя этого типа отсылает нас к слову “character” (буквально: *символ*), думайте о нём как о числе размером 1 байт.
- Литерал[1] `'x'` соответствует ASCII-коду символа "x". Его тип это `int`, не `char`, но стандарт гарантирует, что он уместится в `char`.

- ```
  char number = 5;
  ```

- ```
  char symbol_code = 'x';

  char null_terminator = '\0';
  ```

Тип `int`

- Может быть знаковым или беззнаковым, по умолчанию знаковый.
- Можно просто писать signed, unsigned. Например:

- ```
  signed i = -10;

  unsigned j = 100;
  ```

- Может быть помечен `short` (2 байта), `long` (4 байта на 32-битных архитектурах, 8 байт на AMD64).
- Большинство компиляторов поддерживают `long long`, стандарт описывает `long long` начиная с C99.
- В зависимости от архитектуры, размер `int` меняется. Изначально задумывалось, чтобы размер `int` соответствовал разрядности архитектуры. Около 30 лет назад доминирующими были 16-разрядные архитектуры, затем 32-разрядные, относительно недавно мы перешли на 64-разрядные. В 32-разрядную эпоху нерадивые программисты написали множество программ из расчёта на то, что `int` имеет размер 4 байта и всегда будет таким (хотя если бы они знали историю или хотя бы прочитали стандарт они бы не стали так делать). Программы просто переставали работать если поменять размер `int` на 8 байт, поэтому чтобы не переписывать огромное количество приложений размер `int` как правило по прежнему 4 байта, хотя архитектуры у нас уже 64-битные.
- Важно: все численные литералы имеют по-умолчанию формат `int`. С помощью суффиксов можно это изменить:

- ```
  42    // int
  ```

- ```
  42L   // long int
  ```

- ```
  33UL  // unsigned long int
  ```

  Почему это важно? Для примера возьмём операцию побитового сдвига влево. Запись `3 << 5` означает число 3 которое в двоичной форме сдвинули на 5 бит влево, отбросив 5 старших разрядов и заполнив младшие разряды нулями: было 11_211_2, стало $1100000\_21100000_2$.

- Чему равно значение выражения `1 << 48`? Сдвинем единицу на 48 разрядов влево и получим... ноль. Тип единицы по умолчанию `int`, поэтому всего разрядов в этом числе 32. Однако добавим суффикс L: `1L << 48`, и значением этого выражения будет 2^{48}.

Тип `long`

- На 64-разрядных архитектурах тип `long` как правило занимает столько же места, сколько и `int`; на Windows, правда, 4 байта. Об этом поговорим в связи с моделями данных.

Тип `long long`

- Размер 8 байт

Типы с плавающей точкой

Также есть типы `float` (число с плавающей точкой размером 4 байт) и `double` (число с плавающей точкой размером 8 байт). Литералы с суффиксом f считаются числами типа `float`, например, 1.0f; без него – типа `double`, например, 3.1415.

У арифметики чисел с плавающей точкой много странностей и особенностей, и мы не будем в них углубляться. Они примерно совпадают в разных языках, поэтому их лучше изучить отдельно. Для примеров, однако, пользоваться этими типами чисел мы будем.

[1] Так называется непосредственно заданное значение данных. Например, `x` это не литерал, `4 + 99` не литерал, а `"hello"` – строковый литерал, а `42` – численный литерал.

Пример работы со стандартом: каким может быть размер данных типа `short int`?

Тип данных `short int` может занимать разное количество байт, как и `int`. Стандарт языка говорит, что `short int` должен вмещать **как минимум** числа из диапазона $[-(2^{15}-1), (2^{15}-1)]$, но на деле размер может быть и большим[1].

Как мы об этом узнали? Открываем стандарт ISO/IEC 9899:2017, ищем по тексту `short int` и находим секцию
"5.2.4.2.1 Sizes of integer types <limits.h>". Там рассказывается о заголовочном файле `limits.h`, в котором содержится список макроопределений: директив `#define <имя> <значение>`. Они используются чтобы дать программисту возможность в коде узнать границы на значения, которые могут хранить типы, и их размеры. С разными компиляторами для разных платформ поставляются разные версии этого файла, поэтому при компиляции одной и той же программы для разных платформ значения этих

макроопределений могут быть разными. Для `short int` в файле `limits.h` указаны две константы:

— minimum value for an object of type **short int**

SHRT_MIN	-32767 // $-(2^{15}-1)$

— maximum value for an object of type **short int**

SHRT_MAX	+32767 // $2^{15}-1$

Значит, в этом файле могут быть такие определения:

```
#define SHRT_MIN -32767

#define SHRT_MAX +32767
```

Могут, но не обязаны – это граничные значения. Могут быть и, например:

```
#define SHRT_MIN -32767

// можно сделать больше максимального

#define SHRT_MAX +65536
```

[1] Обратите внимание, что этот диапазон симметричный, а обычно нижняя граница диапазона на единицу больше по модулю: -256...255, -65536...65535 и т.д.

Новые целочисленные типы (платформонезависимые)

Надеюсь, вам уже стало не по себе от хаоса целочисленных типов, присутствующих в C. К счастью, когда вы пишете ваши собственные программы, вы можете использовать появившиеся в стандарте C99 *платформонезависимые* типы данных. Они определены в заголовочном файле `stdint.h` и выглядят единообразно:

- Знаковые типы:
  - `int8_t`
  - `int16_t`
  - `int32_t`
  - `int64_t`
- Беззнаковые типы:
  - `uint8_t`
  - `uint16_t`
  - `uint32_t`
  - `uint64_t`

В этих типах явно указан их размер и знаковость, что снимает все неоднозначности. Прежде чем мы научимся использовать их в функциях `printf` и `scanf`, нужно уточнить одну синтаксическую особенность языка, связанную со строковыми литералами.

Конкатенация строковых литералов

Написав два строковых литерала подряд через пробел мы на самом деле их сольём в один, вот так:

```
"hello" "world" // то же самое, что и:

"helloworld"
```

Благодаря этому мы можем вставлять в одни строки другие, которые являются частью макроопределения:

```
#define WORLD "world"



"hello, " WORLD "!"// то же самое, что и:

"hello, world!"
```

Ввод и вывод с платформонезависимыми типами

Теперь мы можем осознанно использовать полезный механизм, предоставленный заголовочным файлом `inttypes.h`, чтобы использовать новые целочисленные типы в `printf` и `scanf`. Для каждого такого типа нужен собственный спецификатор вывода (для `printf`, как `"%d"` для `int`) и ввода (аналогично для `scanf`). Так же как и с новыми типами нам не потребуется запоминать кучу спецификаторов, но лишь схему формирования макроопределений, которые им соответствуют:

- `PRI` или `SCN` (вывод или ввода)
- Формат: `x` (16-ричный), `u` (беззнаковый), `i` или `d` (знаковый)
- Размер в битах или `PTR` для указателей

Примеры:

```
#include <inttypes.h>

#include <stdio.h>



void f() {

    int64_t i64 = -10;
```

```
    uint64_t u64 = 100;

    printf( "Signed 64-bit integer:   %" PRId64  "\n", i64 );

    printf( "Unsigned 64-bit integer: %" PRIu64  "\n", u64 );

    printf( "Pointer, hexadecimal     %" PRIxPTR "\n", &i64 );



    scanf( "%" SCNd64, &i64 );

    scanf( "%" SCNu64, &u64 );

}
```

Заметьте, что *символы процентов перед спецификаторами нужно указывать явно*.

Файл `stdint.h` включается в файл `inttypes.h`, поэтому его содержимое доступно в этом примере.

С настоящего момента мы полностью переходим на платформонезависимые типы где только возможно. Использование типа `int` будет считаться ошибкой. Также, для их правильного вывода и ввода, необходимо пользоваться спецификаторами начинающимися на PRI (для вывода) или SCN (для ввода).

Помимо вышеперечисленных типов, в заголовочном файле stdint.h также перечислены такие типы, как:

- `int_least8_t`
- `uint_fast32_t`
- `intptr_t`
- `intmax_t`

Прочитайте о них в секции 7.20 стандарта языка.

Псевдонимы для типов

Можно определить для типа произвольное количество псевдонимов с помощью ключевого слова `typedef`. Например, определим псевдоним для чисел `int` под названием `i32`:

```
typedef int i32;
```

Так реализованы и платформонезависимые типы: разные компиляторы под разные платформы определяют псевдоним `int32_t` для разных типов: где-то это просто `int`, где-то `long int` и т.п.

У этого механизма масса применений, и мы будем часто им осознанно пользоваться.

Никогда не снабжайте свои типы суффиксом `_t`, например, `myint_t`. Эти имена считаются зарезервированными; это значит, что в любом будущем стандарте C может появиться тип с произвольным названием и суффиксом `_t`, в том числе и `myint_t`. Тогда в вашей программе окажется два типа `myint_t`: ваш собственный и новый стандартный, а два типа с одинаковым именем – это ошибка компиляции.

Булевый тип

Как мы уже знаем, результатом сравнения двух чисел будет число типа `int` равное 0 или 1. Но почему не `true` или `false`? Причина в том, что раньше в C не было булевого типа, он появился только в версии стандарта C99.

Программистам при этом всегда не хватало булевого типа. Хотя и предполагалось, что можно использовать вместо него `int`, но это вносило путаницу: например, все функции, которые возвращали `int`, делились на две категории:

- одни по смыслу возвращают число;
- другие по смыслу проверяют условие и отвечают "да" или "нет".

Поэтому каждый программист определял свою вариацию этого типа, обычно как псевдоним для `int` под названием `bool`.

В стандарте C99 этот тип появился, но чтобы не сломать совместимость с большим количеством уже использующих "самодельный" тип `bool`, "официальный" булевый тип в C99 называется `_Bool`. Так избегается конфликт имён. Поэтому в программе, где тип `bool` уже определён самим программистом, можно в дальнейшем использовать `_Bool` в процессе миграции на новый тип.

Если же программист сам не определил тип `bool`, то можно подключить стандартный заголовок `stdbool.h` и использовать имя `bool`. По возможности нужно делать именно так.

```
#include <stdbool.h>


bool divisible( uint64_t a, uint64_t b ) {
   return a % b == 0;
}
```

Скорее всего, через несколько итераций стандарта ключевое слово `_Bool` будет объявлено устаревшим, а в дальнейшем его и вовсе удалят; все будут пользоваться `bool`.

На одном из предыдущих слайдов мы просили вас прочитать про такие типы как `int_least8_t` или `uint_fast32_t`. Закономерный вопрос — зачем нам столько типов? Неужели численных типов разных размеров недостаточно?

Одна из главных причин того, что мы расставляем типы переменных — мы хотим сделать программу легче для осмысления, установить дополнительные значимые связи между именем переменной и её смыслом. С опытом вы будете помечать переменные всё большим количеством "ярлыков" и отражать их в типах:

- Одни байтовые переменные хранят коды символов, другие хранят числа без такого смысла.
- Одни числа это величины, другие это идентификаторы объектов и нужны только чтобы их как-то перенумеровать. Например, у работающих программ (процессов) есть идентификаторы.
- Одни переменные хранят метры, другие килограммы; у них разная размерность и по смыслу некорректно их было бы складывать, хотя и то и другое – числа.
- Одни переменные хранят "сырые" данные, полученные от пользователя и ещё не проверенные на корректность, другие уже прошли валидацию и про них мы точно знаем, что, скажем, количество яблок – натуральное число, число корней квадратного уравнения не может быть больше двух и т.д.
- Точный размер одних переменных нам *важен*, на другие мы накладываем только ограничения снизу (например, в `int_least16_t` точно должно умещаться число размера 2 байта, но если компьютеру удобнее работать с числами побольше, компилятор может сопоставить `int_least16_t`, например, 32-битным числам).

По этой же причине в C есть отдельные типы для *длины* и *разницы указателей*.

Тип size_t

Данные типа `size_t` хранят длину чего-то в байтах: размер массива, одного числа, составного типа данных... или количество, которое выражается натуральным числом. Стандарт гарантирует, что `size_t` достаточно большой, чтобы вместить любую *длину массива*.

А вот в `int`, например, *длина многих массивов не умещается*. Числом типа `int` размером 4 байта можно описать длину массива не более 2 Гб (не четыре, как `unsigned int`, потому что это знаковое число!). Размер практически любого фильма больше 2Гб.
Напротив, `size_t` на распространённых платформах это беззнаковое восьмибайтовое число, чего достаточно, чтобы записать точную длину файла размером порядка 18 миллиардов гигабайт (пока что нам хватает).

Поэтому нельзя использовать `int` как индекс в массиве или как размер массива. Весь язык поощряет использование `size_t` для количества элементов или размера области памяти. Оператор `sizeof` тоже возвращает число типа `size_t`, не `int`; стандартные библиотечные функции используют `size_t`, например:

- `strlen` считает длину строки.

```
size_t strlen(char *s);
```

- `strncpy` копирует строку в буфер, при этом он дополнительно принимает ограничения по размеру, чтобы не записать в буфер слишком много байт.

```
char* strncpy ( char* destination, char* source, size_t num );
```

А если массив маленький, и его размер точно уместится в `int`? Даже в этом случае есть проблемы:

```
char* hello = "Hello, world!";

size_t sz = strlen( hello );


//проблема тут

for( int i = 0; i < sz; i = i + 1 ) {

    ...

}
```

На каждой итерации происходит сравнение `i` и `sz`. Но эти числа имеют разный формат (4 байта, знаковое и 8 байт, беззнаковое). Для их сравнения на уровне машинных инструкций придётся добавить дополнительные команды по конвертации их в единый формат, что плохо для производительности.

Кроме того, если нужна производительность, то важно использовать беззнаковые числа для индексов в битовых массивах. В таком случае, чтобы получить доступ к биту по номеру *n* нужно сначала найти в массиве байт под номером *n/8*, затем найти в нём бит под номером *n%8*. Деление на восемь оптимизируется в побитовые сдвиги вправо, но только для беззнаковых чисел! Сдвиги при этом намного быстрее деления.

Тип ptrdiff_t

Мы говорили о том, что функции, работающие с массивами, используют один из двух подходов:

- принимают адрес начала массива и адрес сразу после последнего элемента;
- принимают адрес начала массива и его размер (как мы теперь знаем, размер должен иметь тип `size_t`).

Как правило всё же применяют второй способ, и сейчас мы поймём, в чём неудобство работы с массивом как с парой указателей.

Когда мы вычитаем два указателя, тип результата – это знаковое число `ptrdiff_t`. Его значение, конечно, определено только тогда, когда указатели ссылаются внутрь одного массива. Кроме того, учтите, что *не любая разность указателей в него умещается*. Легко придумать пример для этого:

- пускай `size_t` и `ptrdiff_t` занимают по 32 бита, т.е. 4 байт.
- тогда диапазон значений `size_t` от [0; 2^{32}-1][0;$2_{32}-1$], а диапазон значений ptrdiff_t [-2^{31} ; 2^{31}-1][-2_{31};$2_{31}-1$]
- возьмём размер массива 2^{31} + $12_{31}+1$, тогда разница указателя на последний элемент и указателя на первый элемент не попадает в диапазон типа ptrdiff_t.

В параграфе 6.5.6/9 стандарта приводится такое описание:

When two pointers are subtracted, both shall point to elements of the same array object,... the result is the difference of the subscripts of the two array elements. The size of the result is implementation-defined, and its type (a signed integer type) is `ptrdiff_t` *defined in the* `<stddef.h>` *header.* ***If the result is not representable in an object of that type, the behavior is undefined.*** *In other words, if the expressions* `P` *and* `Q` *point to, respectively, the* `i`*-th and* `j`*-th elements of an array object, the expression* `(P)-(Q)` *has the value* `i-j` ***provided the value fits in an object of type*** `ptrdiff_t`.

Это значит, что нет гарантий, что результат вычитания указателей помещается в ptrdiff_t.

По этой причине мы предпочитаем работать с указателем на начало массива и смещением относительно его начала.

Создайте массив правильного типа и определите функцию в соответствии с комментариями.

Можно считать, что все числа в массиве не превышают 4294900000 и неотрицательны.

1task – program

Приведение типов (Type Casting)

Возьмём функцию, принимающую `int64_t`:

```
int64_t square( int64_t x ) { return x * x; }
```

Мы можем вызвать её с аргументом типа `int8_t`:

```
int8_t a = 42;
```

```
printf("%" PRId64 , square( a ) );
```

Интуитивно такая операция имеет смысл, ведь маленькое число надо всего лишь "расширить" до большого. Однако с точки зрения компилятора мы использовали выражение типа `int8_t` вместо выражения типа `int64_t`; для произвольных двух типов так делать нельзя. Например, компилятор откажется принимать число вместо строки (`char*`).

Для некоторых пар типов, однако, мы делаем исключение – в таком случае мы говорим, что в коде происходит *неявное преобразование типов* (implicit conversion) в том месте, где мы используем "неподходящий" тип.

В общем случае можно сказать, что неявное преобразование из типа A в тип B – это некоторая функция; компилятор замечает "ошибки" типизации в выражениях, где вместо выражения типа B стоит выражение типа A, и исправляет их, вставляя вызов этой функции.

Вот ещё один пример: арифметика с дробными и целыми числами.

```
/*
 Это число с плавающей запятой размером 4 байта
 Суффикс f после числа означает,
 что речь о числе одинарной точности (float).
 Числа двойной точности размером 8 байт называются double.
*/


float f = 1.0f + 3;
```

Целые и действительные числа представляются в компьютере совсем по-разному и складывать числа в разных форматах просто так нельзя. Сначала их нужно привести к единому формату, затем производить сложение. Поэтому этот код на самом деле выглядит для компилятора так:

```
float f = 1.0f + int_to_float(3);
```

У компилятора заготовлена функция, которую мы здесь назвали `int_to_float`; прежде, чем производить сложение чисел, программа выполнит код конвертации и будет использовать не целое число `3`, а число с плавающей точкой `3.0f`.

Также мы можем взять одно выражение и явно сконвертировать к другому типу, если такая конверсия имеет смысл (нет смысла, например, конвертировать число в функцию). Для этого мы пишем новый тип в скобках перед выражением (а само выражение тоже ставим в скобки):

```
/*
 Это число с плавающей запятой размером 4 байта
```

```
 Суффикс f после числа означает,

 что речь о числе одинарной точности (float).

 Числа двойной точности размером 8 байт называются double.

*/

float f = 10.3f ;



// Число округляется к нулю, просто отбрасываем дробную часть.

int64_t i = (int64_t)(f + 4.9f);



// Для простых выражений можно опустить скобки

int64_t i = (int64_t)f;
```

Это *явное* (explicit) *приведение* типов: мы явно указываем, какое выражение и в какой тип хотим конвертировать. Разумеется, если мы конвертируем данные из какой-то переменной в другой тип, тип самой переменной при этом не меняется, и значение тоже.

Неявное преобразование: если определить для типа псевдоним с помощью `typedef`, между типом и псевдонимом появляется неявное преобразование в две стороны.

В дальнейшем мы будем показывать, где в языке случаются неявные преобразования типов и где можно вставлять явные.

Нестрогая типизация

Говорят, что в языке C *нестрогая типизация*. Это означает, что существует достаточно много неявных преобразований типов.

Напротив, если бы количество всего неявного было сведено к минимуму, то мы бы говорили о строгой типизации.
Например, в OCaml есть два оператора сложения: `+` для целых чисел и `+.` для действительных. Если во второй передать целое число, компилятор сообщит об ошибке:

```
1.0 +. 2.0

правильно



1.0 +. 2

неправильно
```

Integer promotion и неявные преобразования чисел

Первые неявные преобразования, которые мы рассмотрим, происходят между числами разного формата. Когда вместо числа типа A мы используем число типа B, компилятор будет действовать по следующему алгоритму.

1. Если B меньшей разрядности, чем int, то конвертируем B -> `int` (это называется integer promotion).
   Если B знаковый, то мы преобразуем B -> `signed int`, а если беззнаковый, то B -> `unsigned int`.
2. Если полученный тип отличается от A, то мы двигаемся по лестнице типов вверх до типа A.

```
int       -> unsigned int       ->

long      -> unsigned long      ->

long long -> unsigned long long ->

float     -> double             ->

long double
```

Первый пункт в конверсии очень важен, так как переполнения в меньших типах случаются иначе, нежели в int.

Для примера, допустим, что `sizeof(int) == 4`. Возьмём два числа типа `unsigned char` и сложим их так, чтобы результат не поместился в диапазон $[0;255][0;255]$:

```
unsigned char  x = 150;

unsigned char  y = 150;

unsigned char r = x + y;   /*  = 44 */
```

Значение, ожидаемо, усечётся до 44, т.к. 300 % 256 = 44.

На самом деле, контринтуитивно, в этом коде происходит следующее: операнды конвертируются в тип `unsigned int`, складываются, а затем результат усекается чтобы вместиться в `unsigned char`.

```
unsigned char r = (unsigned char) (

                                    (unsigned int) x

                                  + (unsigned int) y

                                  );
```

На уровне машинных инструкций такая логика мотивирована тем, что `int` это "родной" для машины формат данных, и чтобы производить арифметику с данными *меньших* форматов она всё равно сначала приведёт их к удобному для себя формату `int`.

Однако по этому же принципу если поместить результат в `unsigned int`, то его значение не будет усекаться:

```
unsigned int r = x + y;  /*  = 300 */


// На самом деле, означает:

unsigned int r =  (unsigned int) x

                + (unsigned int) y;
```

С другой стороны, начиная с типа `int`, всё происходит проще и логичнее:

```
/* предполагаем sizeof(int) == 4 */

unsigned int x = 3000000000;

unsigned int y = x;


/* x + y не умещается в unsigned int */

/* r_int == 1705032704 == 6000000000 % (2^32) */

unsigned int r_int = x + y;



/* хотя результат мы кладём в long,

от переполнения нас это не спасает */


unsigned long r_long = x + y ;   /*  то же самое: 1705032704 */


/* а так всё хорошо: 6000000000 */

unsigned long r_long = (unsigned long) x + (unsigned long) y ;
```

Всегда лучше написать не кратко, но понятно и однозначно:

```
long x = (long)a + (long)b + (long)c ;
```

Даже люди, которые программируют только на C каждый день годами, могут забыть что-то про правила конвертации чисел, поэтому, особенно в контексте программирования в индустрии, стремитесь писать всё максимально явно.

Типы указателей и void*

Мы знаем, что из любого типа `T` можно образовать тип-указатель `T*`. В данных типа `T*` хранятся адреса данных типа `T`.

Существует специальный тип `void*`, "указатель на что угодно". В переменную типа `void*` можно записывать любой указатель.

Неявное приведение. *Любой указатель можно неявно привести к типу* `void*`*, т.е. использовать как значение переменной или аргумента типа* `void*` *без явной конверсии.*

Все указатели имеют одинаковый размер в рамках одной архитектуры, в случае вашего компьютера это, скорее всего, 8 байт.

В общем случае указатели *не обязательно являются числами*. Например, еще 30 лет назад мы пользовались другими ПК, на которых указатели могли иметь форму из пары чисел: базового адреса и смещения относительно него . Однако стандарт требует наличия алгоритма перевода между числами и указателями в две стороны, что позволяет, например, печатать указатели с помощью `printf`:

```
// По стандарту перед печатью надо приводить указатель явно к типу void*

char x;

char* i = &x;

printf("%p", (void*) &i);
```

Указатель `p` типа `void*` нельзя разадресовать, т.е. обратиться по нему. Мы не знаем, на данные какого типа он указывает, поэтому и тип выражения `*p` неясен. Необходимо сначала сконвертировать его в другой тип указателя, а потом обратиться по нему:

```
int64_t x = 0;



void* p = &x;
```

```
// ошибка

int64_t y = *p;



// правильно

int64_t y = * ((int64_t*) p);
```

Разрешается явно преобразовывать тип указателя к другому типу указателя.

Массивы

Нам часто требуются контейнеры – структуры данных, в которые можно складывать данные одного типа, а затем доставать их оттуда поодиночке или пачками. Массив это один из таких контейнеров, обладающий следующими свойствами:

- Сложно добавлять элементы.
- Сложно удалять элементы.
- Легко обращаться к элементу по его индексу.
- Легко считать количество элементов.
- Легко последовательно перечислять элементы.

Примеры других контейнеров вы могли видеть в языках более высокого уровня: хэш-таблица, связный список, двусвязный список, дерево, куча и пр. Например, для связного списка:

- Легко добавлять элементы.
- Легко удалять элементы.
- Сложно обращаться к элементу по его индексу.
- Сложно считать количество элементов.
- Легко последовательно перечислять элементы.

Каждый контейнер делает одни операции легкими и удобными, но другие – сложными и громоздкими. Не бывает идеального контейнера, где все операции максимально быстры и удобны. Мы выбираем контейнер по ситуации, в зависимости от того, какие операции чаще используем.

На уровне встроенных конструкций языка С поддерживает только один контейнер -- массивы. Массивом в С может считаться вообще любой набор данных одного типа, которые последовательно лежат в памяти. Типы массивов не совпадают с указателями, но:

Неявное преобразование. *Тип-массив неявно преобразуется в указатель.*

Поэтому мы уже использовали имя массива как указатель на его начало.

Массивы как аргументы функций

Рассмотрим такой пример.

```
bool starts_with_0(int64_t array[], size_t sz ) {
```

```
    if ( sz == 0 ) return false;

    return array[0] == 0 ;

}
```

Обратите внимание на то, что функция принимает массив. Значит ли это, что массив в функцию копируется целиком?

На самом деле, мы можем переписать функцию эквивалентно так, заменив массив на указатель:

```
bool starts_with_0(int64_t* array, size_t sz ) {

    if ( sz == 0 ) return false;

    return array[0] == 0 ;

}
```

Типы-массивы в аргументах функции – это особый случай, компилятор переводит их в типы-указатели.

Синтаксически вы можете указать в квадратных скобках размерность массива, но **она ничего не означает -- это комментарий.**

```
// добавив 42 мы не изменим ничего

bool starts_with_0(int64_t array[42], size_t sz ) {

    if ( sz == 0 ) return false;

    return array[0] == 0 ;

}
```

Начиная с версии C99 есть специальный синтаксис чтобы сформулировать ограничение "на вход подаётся массив в котором как минимум N элементов". Это может помочь компилятору лучше оптимизировать код. Но оно тоже не проверяется...

```
// в этом массиве как минимум 1 элемент

bool starts_with_0(int64_t array[static 1], size_t sz ) {

    return array[0] == 0 ;

}
```

Константные типы

Для каждого типа `T` мы можем образовать от него неизменяемый тип `const T` (или `T const`, что эквивалентно). Переменные таких типов нельзя изменять напрямую, т.е. каким значением мы их инициализировали, такими они и останутся.

```
int a;

a = 42 ;       /* ok */

...

const int a; /* compilation error */

...

const int a = 42; /* ok */

a = 99;  /* compilation error, нельзя изменять константы */

int const a = 42;  /* ok */

const int b = 99;  /* ok, const int === int const */
```

Интересно заметить, что в случае с указателями `const` может относиться или к тому, что указывает (указатель) или к тому, на что указывает:

- `int const* x` значит "изменяемый указатель на неизменяемый `int`". Нельзя после этого написать `*x = 10`, но изменять сам `x` можно.
  То же самое: `const int* x`.
- `int* const x = &y` значит "неизменяемый указатель на изменяемый int". Можно менять значение по указателю, но перенаправлять указатель на другую ячейку памяти -- нет.
- `int const* const x = &y` значит неизменяемый указатель на неизменяемые данные ; is “an immutable

Правило. `const` *слева от звёздочки защищает то, на что мы указываем; справа -- сам указатель.*

Разумеется, тип `void const*` тоже существует.

Как пользоваться неизменяемыми типами

Если вы хотите сделать что-то изменяемым, это должно быть осознанное решение, а не опция по-умолчанию.

Параметры функций

Возьмём функцию, которая считает количество положительных чисел в массиве.

```
size_t count_pos( int64_t* array, size_t sz ) {
```

```
    size_t count = 0;

    for( size_t i = 0; i < sz; i = i + 1 ) {

        if ( array[i]  > 0 ) { count = count + 1; }

    }

    return count;

}
```

Очевидно, что массив `array` эта функция менять не должна, а если мы это случайно сделаем, это точно ошибка. Поэтому лучше защитить массив от изменения:

```
size_t count_pos( const int64_t* array, size_t sz ) {

    size_t count = 0;

    for( size_t i = 0; i < sz; i = i + 1 ) {

        if ( array[i]  > 0 ) { count = count + 1; }

    }

    return count;

}
```

Представьте, что вы другой программист и смотрите на сигнатуру этой функции. Сравните два варианта:

```
size_t count_pos( int64_t* array, size_t sz );

size_t count_pos( const int64_t* array, size_t sz );
```

По второму варианту сразу видно, что массив изменяться не будет. И это очень полезная информация.

Константы

Традиционно в C глобальные константы определялись с помощью макропроцессора:

```
#define CONST_NAME 42
```

Во многих случаях лучше определять константы как глобальные неизменяемые переменные; например, потому что они типизированы, а литералы – нет. В некоторых ситуациях это не сработает – например, для размеров массивов.

Неявное приведение. *Любой указатель T* неявно приводится к типу T const*; наоборот нельзя, иначе мы нарушим обещание не менять неизменяемые данные.*

Как попытаться обмануть const

Теоретически можно обойти защиту const, обратившись к ней по адресу:

```
const int64_t x = 10;

*( (int64_t*)   &x ) = 30;  // из const int64_t* в int64_t*

printf( "%" PRId64 "\n", x );
```

Стандарт запрещает это действие, но такая программа скомпилируется и (если повезёт) будет вести себя так, как вы предполагаете.

Итог

Вы должны выработать привычку делать неизменямым всё, что только возможно:

- Локальные переменные с промежуточными значениями, которые вы создаёте для удобства.
- Параметры функций, которые являются указателями.
- Глобальные переменные можно использовать только неизменяемые, за редкими исключениями (в контексте многопоточности).

Это помогает отлавливать ошибки и даёт компилятору больше информации, чтобы он мог оптимизировать ваш код и он быстрее работал.

Напишите функцию, которая считает количество нулевых байтов в массиве. Будьте внимательны:

- Массив может быть *любого* типа; если он содержит, например, числа размером 4 байта, то в каждом числе нулевых байтов может быть несколько.
- Функция должна корректно работать с *неизменяемыми* массивами любого типа, ведь она не изменяет их содержимое.
- На вход функции подается не количество элементов, а размер массива в байтах.

2task – program