Типы функций

Функции в C тоже имеют типы, как и более привычные нам данные. Тип функции это её сигнатура: комбинация типов параметров и типа возвращаемого значения. Описываются такие типы не очень привычно:

```
// ftype -- тип функции, возвращающей int, принимающей два аргумента

// типов const char* и float

typedef int ftype(const char*, float);
```

Описание функционального типа очень похоже на объявление функции. Сравните:

```
// Тип функции ftype



typedef int   ftype   (const char*, float);



// Объявление функции myfunction с такой же сигнатурой

// Можно сказать, что её тип совместим с ftype



        int myfunction(const char*, float);
```

Псевдоним типа функции можно использовать для объявлений функций с такой сигнатурой. Синтаксически это похоже на объявление переменных:

```
typedef void action(const char*);



action print;

action log;

action send;



// то же самое, что и:



void print(const char*);
```

```
void log(const char*);

void send(const char*);
```

К сожалению, функциональные типы в C пишутся достаточно громоздко и их тяжело читать. Для удобства мы будем использовать нотацию со стрелочкой, вот так:

- `void -> void`
  Функция, ничего не принимающая и не возвращающая
- `(int64_t, int64_t) -> void`
  Функция, принимающая два числа типа `int64_t` и ничего не возвращающая
- `const int64_t* -> float`
  Функция, принимающая один аргумент типа `const int64_t*` и возвращающая `float`.

Чаще всего функциональные типы используются чтобы создавать указатели на функции.

Указатели на функции

Функции хранятся в памяти в виде закодированных машинных инструкций, поэтому у них есть адреса. Эти адреса можно использовать для вызова функций, а также записывать в переменные, массивы, или передавать в другие функции в качестве аргумента.

В этом примере мы создаём указатель на функцию типа `int64_t -> int64_t` и присваиваем в него адрес функции `square`. Затем мы вызываем функцию по этому указателю, а не по её имени.

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


int main() {

    int64_t array[] = { 1, 2, 3, 4, 5 };

    const size_t count = sizeof(array) / sizeof(array[0]);


    // указатель на функцию int64_t -> int64_t

    int64_t (*mapper)(int64_t) = &square;
```

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

        array[i] = (*mapper)( array[i] );

    }



    return 0;

}
```

Обратите внимание на скобки вокруг имени `mapper`. Без них было бы непонятно, что звёздочка относится к имени указателя `mapper`, а не к возвращаемому типу. Иными словами, неясно было бы, что это:

- объявление функции типа `int64_t -> int64_t*`
- или указатель на функцию типа `int64_t ->  int64_t` (мы бы хотели этот вариант).

Переменная типа указателя на функцию позволяет вызывать через себя то одну функцию, то другую. Для этого достаточно изменить её значение:

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

int64_t cube  ( int64_t x ) { return x * x * x; }


int main() {


    int64_t array[] = { 1, 2, 3, 4, 5 };


    const size_t count = sizeof(array) / sizeof(array[0]);



    // int64_t -> int64_t

    int64_t (*mapper)(int64_t) = &square;
```

```
    // применение функции на которую указывает mapper
    // ко всем элементам массива
    for( size_t i = 0; i < count; i = i + 1 ) {
        array[i] = (*mapper)( array[i] );
    }


    // Заменим значение mapper и выполним тот же код

    mapper = &cube;

    for( size_t i = 0; i < count; i = i + 1 ) {
        array[i] = (*mapper)( array[i] );
    }

    return 0;
}
```

С помощью `typedef` определите тип `ftype` функции , которая принимает два аргумента:

- `const float*`
- (указатель на) функцию, принимающую `struct array` и `int64_t` и возвращающую `size_t`

и возвращает `char`.

Ваше определение должно позволять написать:

```
ftype* fptr = f;
```

где `f` это функция с подходящей сигнатурой.

1task – program

Применение: функции высшего порядка

Указатели на функции это мощный инструмент написания краткого и выразительного кода.

Вернёмся к примеру с применением двух разных функций к каждому элементу массива. Можно думать об этом коде так: каждый элемент массива мы возводим в квадрат и удваиваем. Другими словами, мы хотим уметь "к каждому элементу массива применить какую-то трансформацию", при этом трансформация может быть "возвести в квадрат" или "удвоить". Мы абстрагируемся от того, какая именно это будет трансформация; каждый раз она может быть разной. Чтобы минимизировать количество кода:

- Опишем логику применения произвольной трансформации *T* к каждому элементу массива как отдельную функцию `map`; *T* будет её параметром.
- Оформим каждую интересующую нас трансформацию в виде отдельной функции
- Для применения трансформации к каждому элементу массива передадим в map адрес функции, реализующей её.

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

int64_t cube  ( int64_t x ) { return x * x * x; }




// Параметр-указатель на функцию можно писать без звёздочки

void map( int64_t* array, size_t count, int64_t T(int64_t) ) {

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

      array[i] = T( array[i] );

   }

}


int main() {


   int64_t array[] = { 1, 2, 3, 4, 5 };


   const size_t count = sizeof(array) / sizeof(array[0]);
```

```
    map( array, count, square );

    map( array, count, cube );



    return 0;

}
```

Мы отделили логику *прохода и какой-то трансформации массива* от *трансформации конкретного элемента массива*; теперь первую часть мы можем переиспользовать. Это уменьшило количество кода и подняло уровень абстракции (например, мы больше не думаем каждый раз про границы массива). Как следствие, нам стало легче отлаживать программу, писать и читать код. Фрагмент программы, выглядящий как "примени к каждому элементу массива функцию", легче читается, чем "заведи счётчик типа `size_t`, считай от 0 до размера массива - 1 ... "

Компиляторы умеют встраивать функции типа `map` в месте вызова так, чтобы сгенерированный машинный код был идентичен обычному циклу с применением трансформаций к элементам массива. То есть введение дополнительных функций и их вызов может вообще никак не сказаться на сгенерированном машинном коде.

Функции, которые принимают другие функции в качестве аргумента, называются *функциями высшего порядка (higher order functions)*. В следующем уроке мы реализуем несколько полезных и часто используемых функций высшего порядка для списков: `map`, `fold`, `foreach` и другие.

Преобразования указателей на функции

void* и указатели на функции

Внимание *по стандарту языка нельзя использовать указатели на данные как указатели на функции, и наоборот.*

Мы уже знаем про тип `void*`, который обозначает "указатель на что угодно". По стандарту языка C это "что угодно" может быть *только типом объекта*. Здесь *объект* понимается **не в смысле объектно-ориентированного программирования** – стандарт называет объектом любые данные, которая хранится в памяти абстрактного вычислителя: числа, объединения, строчки, структуры и перечисления, массивы... но не функции. Поэтому тип `void*` по стандарту нельзя преобразовывать в тип указателя на функцию и наоборот[1].

Типы указателей на функции, однако, можно преобразовывать друг к другу. При этом сколько бы мы между ними не делали конвертаций, если в конце привести тип указателя так, чтобы он соответствовал сигнатуре функции, на которую он указывает, то мы сможем корректно вызвать эту функцию по указателю.

```
char f(char x) { return 'a' + x; }
```

```
...

char (*p1)(char) = f;


// преобразуем в const int64_t* -> void

// обязательно явно конвертировать

void (*p2)(const int64_t*) = (void (*)(const int64_t*))(p1);

//                           |                        |  в этот тип мы
преобразуем p1

// это тип указателя на функцию, которая принимает const int64 t* и ничего не
возвращает


// преобразуем обратно в char -> char

char (*p3)(char)  = (char (*)(char))(p2);



// вызовем f через p2

p3(2);  // корректно т.к. тип p3 соответствует типу f, вернёт 'c'


...
```

Если же это преобразование обратно не сделать, то не гарантируется, что вызов будет совершён правильно.

```
void  f(void* x)  { }


...

void (*p1)(void*) = f;


void (*p2)(int64_t*) = (void (*)(int64_t*))(p1);
```

```
p2( NULL );  // некорректно!
             // тип f: void* -> void
             // p2 указывает на: int64_t* -> void
```

Мы увидим важность этого на следующем слайде когда поговорим про функцию `qsort` и обобщённые сортировки.

[1] Однако стандарт POSIX требует, чтобы указатели на функции можно было записывать в `void*` а затем преобразовывать обратно.

Тип функции и тип указателя на функцию

Чаще всего мы используем указатель на функцию чтобы вызвать функцию через него. Для удобства указатель на функцию преобразуется в её собственный тип неявно:

Неявное преобразование. *Следующие две строчки эквивалентны:*

```
array[i] = (*mapper)( array[i] );
array[i] =   mapper ( array[i] );
```

Указатель на функцию прозрачно преобразуется в тип самой функции, что позволяет не разадресовывать его для вызова.

Кроме того, имя функции прозрачно преобразуется в её адрес:

```
int64_t double( int64_t x ) { ... }
...
```

```
int64_t (*mapper)(int64_t);
```

```
// Одно и то же
mapper = &double;
mapper = double;
```

Обобщённые сортировки

Часто на практике нужно отсортировать данные. Сортировка это выстраивание элементов коллекции данных по порядку. При этом порядок можно задавать разный, например:

- можно упорядочить числа по возрастанию или по убыванию;
- можно упорядочить массив пар чисел по второй компоненте пар или по первой;
- можно упорядочить коллекцию записей о пользователях по имени пользователя или по их e-mail адресу, или по любым другим их данным.

Для примера в качестве коллекции элементов возьмём массив. Если мы напишем функции, сортирующие массивы разных данных в любых порядках, эти функции будут различаться в двух аспектах:

- типом элементов массива;
- способом сравнения этих элементов (возьмём любые два элемента; какой из них должен в отсортированном массиве быть ближе к началу?)

Значит, эти функции можно описать как *одну функцию* с двумя дополнительными параметрами:

- один параметр будет содержать информацию о типе элементов массива (достаточно их размера);
- другой параметр будет указателем на функцию, сравнивающую два элемента массива тем или иным способом. Эта функция и задаёт *порядок*, в котором массив сортируется.

Такая обобщённая функция называется *обобщённой сортировкой*. Стандартная библиотека языка C реализует один из алгоритмов обобщённой сортировки под названием quicksort:

```
void qsort(
    void *base,
    size_t nitems,
    size_t size,
    int (*compar)(const void *, const void*)
);
```

Функция `qsort` принимает адрес начала массива `base`, количество элементов `nitems`, размер каждого элемента `size` и функцию-компаратор `compar`. Компаратор заключает в себе часть логики, различающую, скажем, сортировку по возрастанию и сортировку по убыванию. Он сравнивает два элемента по указателям на них и отвечает, какой из них "меньше", а какой – "больше".

Пусть функция-компаратор запущена на аргументах `x` и `y`. Возможны три варианта:

- `x` должен стоять левее `y`; `x` "меньше" `y`, функция возвращает -1
- `x` должен стоять правее `y`; `x` "больше" `y`, функция возвращает +1
- `x` и `y` можно ставить в любом порядке друг относительно друга; они равны, функция возвращает 0.

Обратите внимание, что в качестве компаратора нельзя передавать функцию, сравнивающую, например, два числа: её сигнатура обязана содержать два параметра в точности типа `const void*`.

Вот пример использования `qsort` для сортировки массива чисел типа `int64_t`.

```
// Компаратор

// Важно не терять const

int int64_comparer( const int64_t* x, const int64_t* y ) {

  if (*x > *y) return 1;

  if (*x < *y) return -1;

  return 0;

}



// Адаптер для компаратора

// Предыдущий слайд показывает, почему нельзя передавать

// компаратор типа (const int64_t*, const int64_t*) -> int

// в функцию qsort

int int64_void_comparer(const void* _x, const void* _y ) {

  return int64_comparer(_x, _y );

  // в этой строчке мы используем const void* вместо const int*

  // это правильное неявное преобразование типов

}



...
```

```
int64_t array[] = { 5, 4213, 2381, -1231, 23912309, 3883 };

const size_t count = sizeof(array) / sizeof( array[0] );



// qsort( *base, nitems, size,           (const void *, const void*) -> int )


qsort(    array, count,  sizeof(int64_t), int64_void_comparer );
```

Функция `qsort` принимает указатель на компаратор типа `(const void*, const void*) -> int` и вызывает внутри себя функцию-компаратор по именно такому указателю. Как вы помните, нельзя вызывать функцию по указателю, если её тип не совпадает с типом указателя. Поэтому, хотя нам бы и хотелось ограничиться написанием функции `int64_comparer` типа `(const int64_t*, const int64_t*) -> int`, мы также сделаем для неё адаптер `int64_void_comparer` типа `(const void*, const void*) -> int`.

Используйте функцию `qsort` чтобы отсортировать массив структур по разным полям этих структур.

2task – program

Callbacks

Часто нам удобно думать о работе приложений как о наборе реакций на действия пользователя. Какие события могут произойти и как программа должна на них реагировать?

Один из распространённых примеров это графические приложения. Для их создания мы используем графические библиотеки: GTK, QT, Nuklear, Swing (для Java) и т.д., которые позволяют создавать кнопки, текстовые поля и другие элементы графического интерфейса.

Нам интересно описывать, что происходит, когда пользователь наводит курсор на кнопки, щелкает по ним, вводит текст в поля и т.д. Библиотеки помогают нам в этом, предоставляя возможность описать реакцию на каждое событие. Реакции задаются через передачу в одну функцию (регистратор) адреса другой функции (обработчика), которую нужно вызывать каждый раз при наступлении события – отсюда и название таких функций-обработчиков, "call back". Because the program calls them back when the event happens.

При этом обработчики событий должны принимать в аргументах всю необходимую им информацию. Как правило, это описание события и некий *контекст* в котором оно

происходит. Контекст должен включать в себя все данные, необходимые для обработки события, например:

- слепок состояния программы на момент выполнения события (копии значений важных переменных).
- адреса тех переменных, которые необходимо изменять при наступлении события (чтобы и другие части программы увидели измененённые данные).

Теоретически, можно было бы использовать изменяемые глобальные переменные для передачи данных в обработчики, но мы помним, что изменяемых глобальных переменных следует избегать.

Попробуем сымитировать программу с несколькими кнопками, но без графического интерфейса: для нажатия на кнопку нужно будет ввести её номер. В этой программе две кнопки:

- `0:[Say meow]`, при нажатии выводит `Meow!`
- `1:[Status]`, при нажатии выводит количество нажатий на первую.

```
// контекст для обработчиков событий

// нас интересует, сколько раз мы уже нажали на первую кнопку

struct context { int64_t counter; };



// У кнопки есть имя и обработчик

struct button {

  const char* label;

  void (*handler)( struct button*, struct context* );

};



// Для удобства создадим тип обработчика

typedef void (onclick_handler)(struct button*, struct context*);



// Массив из нескольких кнопок

struct buttons { struct button* array; size_t count; } ;
```

```
// Вызвать событие "Нажатие на кнопку"
void click(const struct button* b, struct context* ctx) {
  b->handler( b, ctx );
}


// Обработчик нажатия на первую кнопку
void print_meow_handler(const struct button* b, struct context* ctx) {
  printf("Meow!\n");
  ctx-> counter = ctx-> counter + 1;
}

// Обработчик нажатия на вторую кнопку
void print_ctx_handler(const struct button* b, struct context* c) {
  printf("Said \"Meow!\" %" PRId64 " times.\n", c->counter );
}

// Показать кнопки
void print_buttons( struct buttons buttons ) {
  for (size_t i = 0; i < buttons.count; i = i + 1 ) {
    printf("%zu : %s \n", i, buttons.array[i].label );
  }
}

// Цикл для ввода номера кнопки и нажатия на неё
```

```
void prompt_click_button( struct buttons buttons, struct context* ctx ) {
  for(;;) {
    print_buttons( buttons );
    printf("Input button index: ");
    const size_t idx = read_size();
    if ( idx >= buttons.count ) { printf("No such button, bye.\n"); break; }
    else { click( buttons.array + idx, ctx ); }
  }
}



int main() {

// Две кнопки
  struct button buttons[] = {
    { "Say Meow" , print_meow_handler },
    { "Status", print_ctx_handler },
  };
// Количество кнопок
  const size_t count = sizeof(buttons) / sizeof(buttons[0]);

// Экземпляр контекста
  struct context ctx = { 0 };

  prompt_click_button( (struct buttons) { buttons, count } , &ctx );
```

```
  return 0;

}
```

Советуем прочитать

[Полнофункциональный I/O реактор на голом Си]

В реальных программах у событий может быть множество обработчиков. Давайте реализуем такую систему, где можно навесить сколько угодно обработчиков на каждое событие!

Представим, что мы управляем роботом, который может перемещаться вперёд, вниз, влево или вправо. Когда робот начинает ехать в определённом направлении, это событие! В момент события робот вызывает функцию `move(dir)`, где `dir` это направление движения. При этом должны вызваться все обработчики движения.

Мы хотим иметь возможность в любой момент добавить ещё реакций на код, который робот вызовет в этот момент.

Подсказка: массивы указателей на функции для этого не подойдут, так как у массивов фиксированный размер, но можно сделать список указателей!

3task – program