Объединения

В структурном типе поля идут последовательно (возможно, с пропусками). А в объединении все поля начинаются по одному и тому же адресу, накладываясь друг на друга. Например:

```
#include <stdio.h>

#include <stdint.h>


#include <inttypes.h>


union qword {

    int64_t integer;

    int32_t parts[2];

};


int main()

{


   union qword test;

   test.integer = 0xAABBCCDDEEFF1122;


   printf( "%" PRIx32 ", %" PRIx32 "\n", test.parts[0], test.parts[1] );

   return 0;

}
```

Нажмите чтобы запустить пример

Здесь массив из двух чисел типа `int32_t` накладывается на те же адреса в памяти, что и поле типа `int64_t`.

Пространства имён

Можно объявить тип-структуру struct T и псевдоним для какого-то типа T; это не приведёт к конфликту имён типов.

```
struct T { int64_t value; } ;

typedef int64_t T;

// конфликта нет
```

Однако имена структур и объединений находятся в одном пространстве имён: определив структурный тип struct T мы уже не сможем определить тип-объединение union T (и наоборот).

```
struct T { int64_t value; } ;

union  T { int64_t value; } ; // конфликт со struct T
```

Мы можем определять структуры и объединения как части других структур и объединений; это позволяет делать много любопытных вещей. Например, в этом объединении `union pixel` можно обращаться к трём элементам массива `char`'ов по именам или по индексам.

```
struct pixel_explicit {

    uint8_t a;

    uint8_t b;

    uint8_t c

};



union pixel {

   struct pixel_explicit named;

   uint8_t at[3];

};



union pixel p;
```

```
p.named.b == p.at[1]; // всегда верно
```

Разумеется, этот код будет работать корректно только в определённых условиях (например, поля в `pixel_explicit` должны лежать одно за другим без пропусков), поэтому хорошенько подумайте, прежде чем такое писать.

Применение объединений

Объединения имеют два применения:

- экономия памяти, когда мы переиспользуем одни и те же адреса то для одних данных, то для других;
- переинтерпретация одних и тех же байтов памяти как данных разных типов;

Стандарт описывает объединения только с точки зрения экономии памяти. Иначе говоря, нам гарантируется лишь возможность записать значение в поле объединения и потом прочитать его из того же поля; записи в другие поля объединения сразу запрещают чтения из первоначального поля.

В общем случае, после записи в любое поле объединения значения остальных полей становятся неопределёнными. Исключение из этого правила это объединение из несколько структур, в начале которых поля одинаковых типов:

```
struct sa {
    char x;
    int64_t y;
    char z;
};

struct sb {
    char x;
    int64_t y;
    int64_t notz;
};

union test {
    struct sa as_sa;
```

```
    struct sb as_sb;
};
...

union test test_instance;
test_instance.as_sa.y = 100;

// оба чтения гарантированно приведут к одному результату
printf("%" PRId64, test_instance.as_sb.y ) ;
printf("%" PRId64, test_instance.as_sa.y ) ;
```

Переинтерпретация данных с помощью объединений считается завязанной на конкретную архитектуру и её представления данных, поэтому пользуйтесь ей с осторожностью.

Анонимные структуры и объединения

Начиная с C11 структуры и объединения могут быть анонимными частями других структур и объединений. Раньше мы писали так:

```
union vec3d {
    struct {
        double x;
        double y;
        double z;
    } named ;
    double raw[3];
};

union vec3d v;
v.named.z = 100;
```

Теперь мы можем не давать имя полю `named`; тогда его поля будут доступны без дополнительной точки:

```
union vec3d {
    struct {
        double x;
        double y;
        double z;
    };
    double raw[3];
};
```

```
union vec3d v;
v.z = 100;
```

Перечисления

Перечисления это численный тип данных на основе `int`[1]. Перечисление задаёт некоторый набор констант, снабжённых именами. Например, светофор может быть в одном из следующих состояний :

```
enum light {
  RED,
  RED_AND_YELLOW,
  YELLOW,
  GREEN,
  NOTHING
};
```

Обычно это означает, что переменная типа `enum light` в программе может принимать только значения из этого списка. Но в принципе C, к сожалению, не запрещает присваивать любые целые числа в такую переменную:

```
enum light mylight = RED;

mylight = 1000;
```

Можно явно указывать, каким числам соответствуют именные константы:

```
enum light {

  RED = 100,

  RED_AND_YELLOW = 923,

  YELLOW = 9,

  GREEN,

  NOTHING

};
```

Часто указывают только значение первого элемента, а последующие в таком случае идут по порядку.

```
enum light {

  RED = 0,

  RED_AND_YELLOW, // 1

  YELLOW,         // 2

  GREEN,          // ...

  NOTHING

};
```

В стандарте общее описание перечислений находится в секции 6.7.2.2.

[1] Точнее, в зависимости от конкретной архитектуры, перечисления реализуются через `unsigned int`, `signed int` или `char`.

Применение перечислений

1. Один из способов задать именованную константу без `#define` -- использовать анонимное перечисление.

```
enum { A = 320 };

// можно создавать массив такой длины!
char array1[A] = {0};

// а массив длины, взятой из глобальной неизменяемой переменной, сделать нельзя:
const int B = 320
char array2[B] = {0}; // ошибка компиляции
```

11. Если сущность в программе может быть в одном из небольшого фиксированного количества состояний, то удобно применять перечисление для описания текущего состояния. Например, если мы реализуем конечный автомат.
12. Предположим, функция копирует файл из одного места в другое. Операция может завершиться успехом или может произойти одна из многочисленных ошибок:
    - не хватает прав для записи,
    - мало места на диске,
    - ошибка чтения из файла и т.д.

    Можно написать функцию копирования так:

```
bool copy_file( const char* from, const char* to ) {

...

пытаемся скопировать файл

...

if (на диске не хватает места)  {
  printf("Не хватает места");
```

```
  return false;
}

if (ошибка чтения)  {
  printf("Ошибка чтения с диска");
  return false;
}

return true;

}
```

В этом коде есть две проблемы:

4. вызывающая `copy_file` функция не может знать, что именно произошло в ней не так. Она может знать только получилось скопировать или нет.

5. логика "скопировать файл" смешивается тут с логикой "вывести сообщение об ошибке". А если программа локализуется на несколько языков, как быть с вкодированными прямо сюда сообщениями об ошибках? Переписывать функцию копирования файла (а потом её заново тестировать и т.д.?)

Можно разделить логику функции `copy_file` на три части: основные действия и выдача ошибок; выбор соответствия ошибок и сообщений об ошибке; показ сообщений.

```
enum copy_file_result {
   CF_OK,
   CF_ERROR_OUT_OF_SPACE,
   CF_ERROR_IO
};

enum copy_file_result
copy_file( const char* from, const char* to ) {
```

```
...
пытаемся скопировать файл
...

if (на диске не хватает места)  {
  return CF_ERROR_OUT_OF_SPACE;
}

if (ошибка чтения)  {
  return CF_ERROR_IO;
}

return CF_OK;
}

const char* const cf_error_messages[] = {
[CF_ERROR_OUT_OF_SPACE] = "Не хватает места",
[CF_ERROR_IO] = "Ошибка чтения с диска"
};


void perform_copy_file(const char* from, const char* to) {
  enum copy_file_result status = copy_file(from, to);
  printf( cf_error_messages[status] );
}
```

Так можно легко модицифировать программу для поддержки разных языков, сделать графическое приложение (в котором `printf` не используется для показа сообщений пользователю); функции остаются маленькими, и логика обработки ошибок может быть сконцентрирована в одном месте, а не смешиваться с логикой копирования файла.

Одно из частых применений перечислений, объединений и структур — работа с переменными, которые могут хранить данные одного из фиксированного набора типов. Например:

```
enum animal_type { AT_CAT, AT_FROG } ;


struct cat {
  const char* name;
  bool        meows_often;
  bool        obeys;
  const char* master;
};


struct frog {
  bool quacks;
  bool venomous;
};


struct animal {
  enum animal_type type;
  union {
    struct cat as_cat;
    struct frog as_frog;
  }
};
```

Как это работает:

- Структура содержит перечисление и объединение из нескольких типов данных;
- Перечисление явно показывает, какие именно данные хранятся в объединении (потому что неявно во время выполенения эта информация нигде не хранится):

```
// Это кошка

(struct animal) { .type = AT_CAT, .as_cat = { "barsik", true, false, "Bors Olegovich" } }

// Это лягушка

(struct animal) { .type = AT_FROG, .as_frog = { false, false } }

void f(struct animal a) {
  if (a.type == AT_CAT) {
    printf("It is a cat");
    if (a.as_cat.meows) printf(", it meows\n");
  }
  else {
    printf("It is a frog");
    if (a.as_frog.venomous) printf(", it is venomous\n");
  }
}
```

Конечно, размер `struct animal` определяется наибольшим из размеров полей объединения. Структура `cat` больше, чем структура `frog`, поэтому когда в объединении хранится лягушка, а не кошка, некоторые байты объединения никак не используются.

Тип-сумма

Создайте тип, который может хранить или целые числа, или указатели на строчки. Мы полагаем, что все строчки выделяются в куче.

Такой тип называется *помеченная сумма* двух типов (в данном случае это типы `int64_t` и `const char*`), а обычное объединение – *непомеченная сумма*. Перечисление `type` это пометка: какой именно из альтернатив равняется объединение? Непомеченное объединение может быть чем угодно, и во время работы программы без такой пометки не существует способа проверить, что именно там лежит. Это прямое следствие одного из [принципов фон Неймана](), по которому данные кодируются в памяти нулями и единицами, и не существует способа отделить код от данных, а данные разных типов друг от друга.

1task – program

Типы это информация о возможных данных

В прошлом задании у нас возникла щекотливая ситуация. Предположим, что `either_int_string` хранит указатель на строку. При этом строка занимает место в памяти *где-то ещё*, и эту память, возможно, необходимо освобождать вручную с помощью `free`.

Вспомним, где вообще могут храниться строки:

- Если строка в стеке, то это локальная переменная какой-то функции `f` и она сама уничтожится при выходе из `f`.
- ```
  void f() {
  ```
- ```
    char string[] = "hello";
  ```

  ```
  }
  ```

  Экземпляр `either_int_string` тогда не должен жить дольше, чем `f`, например, копироваться в ту функцию, которая вызывает `f`. Иначе в нём останется указатель на память, которая при выходе из `f` освобождалась.
  Вряд ли мы хотели бы внутри `either_int_string` хранить указатель на строку в стеке.
- Если строка в глобальной области данных, то это глобальная переменная или строковый литерал:
- ```
  // Глобальные изменяемые данные
  ```
- ```
  char[] hello = "hello";
  ```
-

-
-
- `void f(char*);`
-
- `void g() {`
- `   // Здесь создаётся блок глобальных неизменяемых данных,`
- `   // в котором хранится строка Hi;`
- `   // указатель на него передаётся в функцию f.`
- `   f("Hi");`

  `}`

  Глобальные переменные живут вплоть до завершения программы, и освобождать их вручную не нужно.

- Если строка размещена в куче, память для неё выделяется с помощью `malloc`, и должна быть освобождена вручную вызовом `free`.

Нам бы помогло, если бы мы могли различать указатели на *строки в куче* и указатели на *остальные строки*. Так будет ясно, следует ли вызывать `free` для неё когда-нибудь, или нет. **В идеале при попытке использовать один указатель вместо другого компилятор должен выдать нам ошибку**. Есть несколько способов создать новый тип, но не все нам подходят.

1. Создать псевдоним для типа `char*` с помощью `typedef`.

   ```
   typedef char* heap_string;
   ```

   Как вы помните, при создании псевдонима в программе появляются неявные преобразования в две стороны: вместо `char*` теперь можно использовать `heap_string`, и наоборот. Но хотя бы типы в сигнатурах функций и в полях структур теперь могут нести больше информации:

   ```
   void print_and_deallocate( heap_string s ) {
     ...
   }
   ```

   К сожалению, конверсии как раз дело и портят: компилятор **не запрещает** нам передавать вместо `heap_string` любые данные типа `char*`, хотя наша цель была **предотвратить** такую ситуацию.

2. Сделать структуру из одного поля, в которую упаковать указатель.

```
3. struct heap_string {
4.    char* addr;
5. };
6.
7. void print_and_deallocate( struct heap_string s ) {
8.   ...
   }
```

Теперь мы не можем передавать в функцию `print_and_deallocate` произвольные строчки -- только те, что были явно упакованы в структуру `heap_string`. С точки зрения производительности структура `heap_string` и её поле `addr` полностью эквивалентны и по потреблению памяти, и по тому, как с ними будет производиться работа на уровне машинных инструкций, поэтому не бойтесь создавать столько структурных типов, сколько необходимо. Удобно создать функцию для конвертации `char*` -> `heap_string` (выделить память в куче и скопировать туда строку); а обратное преобразование будет всего лишь обращением к полю `addr`.

Типобезопасные псевдонимы с помощью typedef

В случае, когда новый тип является просто обёрткой над старым, автор курса считает *допустимым* использовать `typedef` совместно с объявлением структурного типа, вот так:

```
typedef struct  {
   char* addr;
} heap_string;

// heap_string это псевдоним для анонимного структурного типа struct { char* addr }

// так мы не создаём структурного типа с именем и можем использовать heap string без ключевого слова struct

// Здесь struct используется как хак для создания псевдонима типа без добавления неявных конверсий

// и никак не означает, что внутри неё -- большая структура данных.
```

```
void print_and_deallocate( heap_string s ) {
  printf(s.addr);
  free( s.addr );
}
```

Так можно помечать данные и другой информацией. Например, пользователь вводит любое число, но программа ожидает число от 5 до 7. После проверки можно упаковать число в структуру:

```
typedef struct {
   int64_t value;
} sanitized_int;


sanitized_int sanitize( int64_t i ) {
  if (5 <= i && i <= 7 ) {
     return (sanitized_int) { .value = i };
  }
  else {
    // аварийное завершение, но можно сделать разумную обработку ошибки
    abort();
  }
}



void do_something( sanitized_int i ) {
  // В этом месте кода есть гарантия, что 5 <= i.value <= 7
  // произвольное число без упаковки мы передать в i не можем

}
```

Еще несколько применений для упакованных по-разному данных одного типа:

- Идентификаторы разных сущностей в программе часто являются целыми числами: ID процесса, файловые дескрипторы и т.д. С помощью структур их можно различать чтобы не передавать в функцию, ожидающую файловый дескриптор, длину в метрах.
- Метры, килограммы, литры, км/ч -- нельзя, скажем, складывать величины разной размерности, хотя все они выражаются числами.

Реализуйте упакованный указатель на строку, которая выделена в куче.

2task – program