

Молчание компилятора – неограниченная свобода действий.

Первая программа

3 деда решили создать ОС и попутно получили язык программирования, который им подходил. Си был сертифицирован Американским Национальным Институтом Стандартов (ANSI). K&R не дала четкой спецификации к компилятору.

Расширение .h говорит о том, что это заголовочный файл. Компилятор копирует коды библиотек и позволяет использовать функции, описанные в этих библиотеках.

Файл "simple.h" пишется в "", а не в <> потому что это не файл из стандартной библиотеки (СБ). Файлы СБ обычно располагаются в папке include

... Я не научился добавлять новые файлы в проект... Надеюсь... в будущем CMake меня спасет) ♥

Паника отменяется! Нужно было правильно вызывать компилятор. Вот так: gcc w.c simple.c Он выдает a.exe и все работает.

Переменные

*Название переменной не может начинаться с числа.

Char – 1 байт - 255

short – 2 байта - 65535

int – 4 байта – 4 млрд.

long – 4 байта – тоже самое

long long – 8 байт. – много.

Если переменная подписана, то центр плеча перемещается на ноль и сумма делится между положительными и отрицательными числами.

sizeof – размер переменной в байтах.

Типы с плавающей точкой:

float – 4 байта. Макс: 3.3e+38

long float – 8 байт

double – 8 байт

long double – 8 байт. Макс: 1,79e+308

Тааакк. Unsigned – это просто переменная в которую записаны числа))). Все просто... Переполнение реально..

Постфиксное обозначение типа

При работе с числами можно с помощью литер явно указывать его тип.

11 – int. 10u – unsigned. 22l или 22L – long. 80.0f – float. Обязательно наличие десятичной точки при записи. 3.0 – double

Шестнадцатеричный и восьмеричный формат: начало с 0x и с 0. Хекс и отк можно спокойно переводить в десятичную систему через %d.

Printf(%d, %x, %o, %u, %lld, %c, %s, %p)

%lld – позволяет отображать лонг лонг числа. (выводит все число)

Экспоненциальная форма числа: 1.25e-22;

▲ После объявления переменной она не приравнивается автоматически к нулю. В ней хранится мусор, который остался в той же области памяти. ! Глобальные переменные обнуляются при инициализации.

Unsigned a, b, c;

a = b = c = 100;

* При выделении памяти можно узнать была она выделена или нет, если прозвонить переменную по if.

Lvalue – то, что слева от “=”. Rvalue – то, что справа

Ввод/вывод

%c – буква, %s – строка, %p – указатель. %% - вывод знака %.

“Выключка не работает и не показана..” %d не работает.

%.6f – вывод float с точностью по знаку “.6”

printf("%.6f\n", 1, a); - вывод с точность, по *. Исп. 2 аргумента.

printf("%020.3f\n", d); - выставляет число нулей.. не знаю зачем. Но они добавляются перед числом.

printf("%*x", 3, a); - вывод числа знаков.. Какой то юзлес.

printf("%.8s", string); - Можно резать строку. – вывод первых 8 элементов.


Объявление строки, как массива символов: char* string = “easy”

* Есть еще суб-спецификаторы, которые изменяют длину выводимого элемента.

Scanf(%d, &dataStorage). Можно сразу же выполнять идентификацию элементов из строки вот так:

scanf(“%d:%d:%d”, &y, &m, &d). Удобненько..

scanf(“%3c”, %buffer) – ограничение на входные данные – первые 3 элемента.

Char s = getch() – любой символ , это 3)

fgets(buffer, 127, stdin) – позволяет считывать строки с пробелами.

* Не забывай, что в scanf передается указатель на область памяти, а не переменная.

При сравнении нужно приводить типы к единому: inr < 0.0f

Поиск остатка от деления такой же, как в JS. 7 % 4 = 3; 8 % 4 = 0

Если добавляешь exit(1) в switch, то добавляй библию <stdlib.h> В case должны быть либо числа, либо символы.

▲ В операторах вычисления происходят слева на право.

Префикс (++i) возвращает новое значение. Постфикс (i++) значение до увеличения.

Постусловие: `do {} while`. Предусловие `while`. * `break` может прибавлять циклы. Нужно использовать чаще.

Массивы

Объявление: `int a[100]`; - массив с 100 элементами `int`. Вот так объявленный массив содержит мусор. Присвоение: `= {1, 2, 3, 4, 5, ...}`

Индекс элемента, это сдвиг на `l*sizeof(тип)` байт от начала. Массив не хранит информацию о своем размере и не проверяет индекс на корректность. Это значит, что можно залезть в ту область памяти, которая дальше массива.

▲ Можно создать массив с нулями так: `int a[10] = {0}`;
Можно объявить массив без указания размера: `int a[] = {1, 2, 3}` – размер 3.

⚡ Можно узнать размер массива вот так: `size(a)/size(int)`

Незнакомая конструкция: `#define SIZE 10u...` глобальный доступ)

`#include <time.h>`

`time_t seconds` – новый тип данных... Выводится в строке, как `long decimal`.

`seconds = time(NULL)` – возвращает секунды с 1970 года.

Генерация случайных чисел требует настройки: `srand(time(NULL))` – инициализация зерна. `rand()` возвращает число от 0 до 32767. При вызове 1ый раз возвращает почти одинаковые значения. При последующих вызовах появляется существенная неопределенность.

Алгоритм перемешивания находит 2 точки и переставляет их. Первая линейная, а вторая от рандомайзера.

Многомерные

`int A[2][3][4]`, где 2 верхних уровня, 3 средних и 4 самых низких.

Можно проводить инициализацию по “удобному”: `int A[2][3] = {1, 2, 3, 4, 5, 6}`. Так числа с 1 до 3 уйдут в первую колонку, а с 4 до 6 во вторую.

```
int a[][2][3] = {1, 2, 3, 34, 5, 6, 7, 8, 9, 10, 11, 12};
```

Такая запись также распределит по структуре 2 – 3 элемента и выделить N первых уровней, которые не указаны.

Многомерный массив – одномерный по структуре. Получается, что можно получить доступ к элементу через его порядковый номер:

`A[i][j] = A[0][i * число столбцов + j]`. Можно легко сделать пузырьковую сортировку.

Как правильно передавать одно и многомерные массивы в функцию???? 📞

```
while (words[i][j]) {  
    j++;  
}  
counter[i] = j;
```

Это проверяет строку на наличие элемента...

Строки

Строка в Си, это массив типа `char`. Последний элемент хранит терминальный символ `'\0'`. Если не добавлять этот элемент, то строка заканчивается с мусором. – Символы произвольной длины до тех пор, пока не будут 0.

Указатели

- переменная, которая хранит адрес области памяти. Указатель, как и переменная, имеет тип. Синтаксис: тип `*имя`. Сущ. 2 оператора: `&` - взятие адреса. `*` - “разыменование”

Используя оператор `*` по отношению к указателю, можно изменить содержимое переменной.

- Создание переменной и указателя.
- Получение адреса переменной и присвоение его указателю: `p = &A`.
- Обращение к указателю через `*p = 200`. Замена значения.

Можно сказать, что есть 2 уровня. Верхний (`*p, A`) здесь в обороте числа и натуральные значения. Нижний уровень (`p, &A`) здесь в обороте информация о памяти. Так связь на нижнем уровне может вызывать на верхнем уровне зависимости. Работает, как при изменении `*p`, так и при изменении `A`.

* Указатели имеют одинаковый размер `size_t`.

Арифметика указателей: Ему нужен тип для того, чтобы корректно работала операция разыменования (получения содержимого по адресу). Если указатель хранит адрес переменной, то необходимо знать сколько байт нужно взять, чтобы получить всю переменную.

- Указатели поддерживают ариф. Операции. `+N` сдвигает указатель вперед на `N*sizeof(тип)` байт
- Можно иметь указатель на начало массива и это позволит далее двигаться по этому массиву, получая доступ до отдельных элементов.

▲ Указатель на массив:

```
int A[10] = {1, 2, 3....};
```

```
int *p;
```

```
p = A // То есть массив сам является указателем
```

```
Но можно и так: p = &A[0]
```

* Можно сделать указатель на указатель и определять его содерж. Памяти.

⚠ Указатель на указатель на указатель используется при работе с массивами).. ЖДЕМ!

**поднимает уровень до числа. Так `p=&B: pp = &p → *pp = p, **pp = B`.

△ С помощью указателя можно залезть внутри типа и посмотреть его содержимое:

Берется указатель на `int` и превращается в указатель тип `char charPtr = (char*)intPtr;`

Затем можно его обходить.. Пользуемся тем, что размер типа `int` = 4 байт, а `char` = 1 байту.

Для определения инициализированного указателя используется NULL макрос.

```
int *p = NULL;
```

Интересное взаимодействие с массивами... То есть повышая *, можно проваливаться до элементов массива.

P = ar, тогда *p = элемент массива, а если сдвигать p на единицу, то можно совершать обход.

☒ При сортировке массива приходится перемещать элементы. Указатели весят меньше и удобнее перемещать их, а не массив. При этом исходный массив не будет изменен. Сортировка будет происходить быстрее!

*** Ебейшие мувы. Так как размер типа char = 1 байту, то с его помощью можно реализовать операцию swap – обмен содержимого 2х переменных... Опять залезаем туда, куда нельзя.

Float a = 0.5f; p1 = (char*)&a; - преобразование N битного адреса к единичному, при его сдвиге можно перейти к следующему биту float. Совершая такой обход можно получить доступ к каждому биту переменной a

Константы и указатели

В Си можно создавать такие указатели, которые могут изменять свое значение, но при этом они могут только читать содержимое памяти и не могут его изменять.

const char* h = &A[2] – не дает изменить значение. Он даже не дает собрать программу.

Чтобы передать строку (одномерный массив) нужно отдать её. В функции получить её, как УКАЗАТЕЛЬ (char *word) и далее привязать новый указатель к массиву (или первому эл. &A[0]). Перемещаясь по указателю имеем доступ к массиву

Int *p = A ===== int *p; p = A; Со звездочкой тоже самое....

Указатели и многомерные массивы:

```
Int A[][3] = {1, 2, 3, 4, 5, ...};
```

```
int *p = A[0]
```

Нужна практика. Без примеров не пойму.

#ifdef #if #ifndef – целый язык программирования внутри макросов.

```
Int a[] = {  
    #include "./array.txt"  
} - ЭТО РАБОТАЕТ
```

Целые типы фиксированного размера

Целые типы фиксированной ширины

Название	Знак	Размер, бит	Размер, байт
int8_t	signed	8	1
uint8_t	unsigned	8	1
int16_t	signed	16	2
uint16_t	unsigned	16	2
int32_t	signed	32	4
uint32_t	unsigned	32	4
int64_t	signed	64	8
uint64_t	unsigned	64	8

Рисунок 1. Таблица размеров фиксированных типов.

Оператор запятая:

$C = (a = 3, b = 4) \rightarrow C = 4$. Запятую можно использовать в условиях, что бы выполнить несколько действий.

Структура приложения на Си

Data состоит из статических и глобальных переменных, которые явно инициализируются значениями. Этот сегмент может быть далее разбит на ro-data (read only data) – сегмент данных только для чтения, и rw-data (read write data) – сегмент данных для чтения и записи. Например, глобальные переменные

BSS-сегмент (block started by symbol) содержит неинициализированные глобальные переменные,

Heap – начинается за BSS. Пространство памяти используемое для динамического выделения.

Стек вызова – область для локальных переменных, которые объявляются в функциях.

И сегмент кода...

Динамическое выделение памяти

Когда размер исп. Пространства не известен, то необходимо динам. Выд. Памяти. – Используется выделение памяти из кучи. Недостатки:

1. Память необходимо очищать вручную.
2. Выделение памяти – дорогостоящая операция.

Malloc – memory allocation. Библиотека stdlib.h

`void * malloc(size_t size);` - есть размеры в байтах. Выделяет size байтов и возвращает указатель на эту память. Указатель нужно привести к нужному типу.

```
Int *p = NULL;
```

```
p = (int*) malloc(100);
```

```
free(p);
```

`p = (int*) malloc(size * sizeof(int)); (int*)` – приведение типов. Пишем такой же тип, как у указателя. !!! ПОСЛЕ ЭТОГО РАБОТАЕМ С УКАЗАТЕЛЕМ, КАК С МАССИВОМ !!!

Динамическое создание 2х мерного массива:

1. Создается массив указателей.
2. Каждому из элементов присваивается адрес нового массива.
3. Удаление в обратном порядке.

```
Float **p = NULL;
p = (float**) malloc(row*sizeof(float*));
for(int l = 0; l < row; l++) {
    p[l] = (float*) malloc(column*sizeof(float));
}
```

На самом верху указатель характеризующий размер malloc массива (**p). Далее идет присваивание верхнего уровня `p = (float**) malloc(row*sizeof(float*))`; так как это массив указателей, то и размеры соответствующие.

Внутри цикла уже происходит присвоение каждому новому указателю массива с нормальным размером float.

Причем `p[l] = (float*) malloc (column*sizeof(float));` - преобразуется к указателю.

✎ Сначала создается массив указателей, а потом каждому указателю присваивается адрес нового массива. Это значит, что можно:

1. Выбирать размер нижнего массива. – Массив строк, каждая из которых имеет разный размер.
2. Работать по отдельности с каждой строкой. Освободить память или менять размер строки.

★ Я хочу создать 3х-мерный массив: создаю указатель 3-го уровня (на указатель указателей). Выделяю память для всех указателей на указатели (предполагаю, что там размер `int*`). Запускаю цикл, где выделяю память на указатели, а потом на числа.

* в 2х мерном массиве был 1 цикл, тут 2 (по вложенности).

`Void* calloc(size, size_t)` – malloc, только без перемножений.

`Void* realloc(ptr, size_t size)` – re-allocation. Позволяет изменять размер ранее выделенной памяти и получает в качестве аргументов старый указатель и новый размер памяти в байтах.

* не забывай про 0 элемент, когда выделяешь память строке.

При проверке на размер нужно четко понимать, что происходит `counter >= size!!!`

★ Вот, что я понял.. Можно передавать в функции указатель на многомерный массив, а там уже работать с ним в виде `pth[0]....`

* все копирования строк лучше делать через `strcpy(buffer, "123")`; Через malloc создаем все массивы и проблем не знаем)

Нужно следить, чтобы в scanf("%s", next) был именно флаг строки.

Функции

Передача адреса:

```
change(int *a) {};
```

```
change(&a);
```

- если изменить переменную внутри функции, то это отразится на внешней переменной.

Для изменения объекта необходимо передавать указатель на него, в данном случае – указатель на указатель.

▲ Если необходимо присвоить указателю область памяти внутри функции, то нужно передать указатель на этот указатель)

В функции можно работать с указателем меньшего уровня, чем тот, что был передан.

Изменение указателя может произойти при обращении на соответствующий уровень.

```
Char *test = init('hi');
```

printf("%s", test); - здесь текст хранится на... нижнем уровне.. Это, потому что тут malloc замешан. Когда используешь malloc, помни, что данные хранятся на нижнем уровне.

Передача массива в качестве аргумента

```
Int *arr; - обращение к элементам arr[0];
```

```
int arr[][5]; - arr[i][j];
```

```
int (*arr)[5]; - arr[i][j];
```

△ itoa(int, strPtr, base) – 1ое: число для преобразования. 2ое – указатель на строку в которой сохранится число, 3ье – система счисления для преобразования.

△ strcpy(words[i], str) – копировалка слова. Работает стабильнее всего на свете.

△ strlen(words[i]) – длина слова.

Любимое – динамический массив:

```
char **words = NULL;
```

```
words = (char**) malloc(size*sizeof(char*));
```

```
getLeng(words, size)
```

```
int *len = NULL;
```

```
len = (int*) malloc(size*sizeof(int));
```

```
Заполнение через len[i]
```

```
return len;
```

Приведение указателей:

```
void *p = NULL;
```

p = &intVar; - установка нижнего уровня.

((int) p) = 20; - установка значения на верхнем уровне.

△ memcpy(*target, *source, n_bytes); - копирует N байтов из источника в новый указатель.

▲ Пустые указатели позволяют создавать функции, которые возвращают и принимают одинаковые параметры ▲ — что это значит?х

*совместно с void указателями можно создавать функции “общего назначения”. Указатели позволяют создавать функции высших порядков.

Создание функции и указателя на неё.

```
Int sum(int a, int b) {return a + b};
```

```
void main () {
```

```
Int (*fptr) (int, int) = NULL;
```

```
fptr = sum; // Символ указателя добавлять не нужно*
```

```
}
```

Указатели на функции нужны для того, что бы делать высшие функции: map.

!!! Не забывай про stdlib.h !!!

Есть вариант производить обход массива через преобразование: `char *ptr = (char*) arr;` и передавать и указатели на новые данные: `(void*)(ptr + i*size);` Причем это промежуточное значение нельзя вывести.. оно в формате void//

★ ЭТО ОЧЕНЬ ВАЖНО, КОГДА РАЗМЕР ЭЛЕМЕНТОВ НЕ ИЗВЕСТЕН, НО НУЖЕН ПЕРЕБОР

Суть того map в том, что там все функции имеют следующий указатель:

`void (*fcn)(void*)`. То есть можно сделать условно универсальный map, который принимает много видов функций.

Внутри этих функций аргумент тоже воспринимается, как `void *data`. Далее происходит преобразование к нужному типу через `(int*)data` и если нужно значение дополнительно доб. Оператор `*`.

* Если требуется создание функций, которые могут есть много чего, то это превращается в такую ебку с аргументами..

`**out = *out[i]` --- это уместно, если передан адрес на указатель, а не сам указатель.

* Перед обращением к указателю через `*` нужно заключить его в скобки:

`(*res)[0]`

Повторение: если обратиться к переменной через адрес, то можно её изменить, даже, если действие происходит внутри функции. – Вне контекста объекта изменения.

*** Можно сделать массив указателей на функции: `float (*menu[4]) (float, float);`

Динамический вариант создания массива функций:

```
menu = (float(**)(float, float)) malloc(4*sizeof(float*)(float, float));
```

“Часто указатели на функции становятся громоздкими и тогда можно ввести новый тип:”

```
typedef float (*operation)(float, float);
```

qsort bsearch – полезный стафф.

Аргументы функций

Если в функцию передается экземпляр структуры, то передается копия этой структуры. – изменения, которые внесет функция не отразятся на внешней элементе. След. Если передать указатель на структуру, то зависимости подтянутся.

Пусть point – указатель на массив:

```
point_t *p = malloc(sizeof(point_t));
```

p->x = 10; - то есть здесь обращение к указателю и выход через него на значение X. -> - это дает выйти на параметры структуры через указатель на структуру.

```
void foo(point_t **point) {  
    (*point) = malloc(sizeof(point_t));  
    (*point) -> x = 100  
}
```

- Здесь в функцию передается указатель на указатель. Тогда поднимаясь на уровень выше через * можно добраться до указателя на struct и затем.

СТРОООКИИ

```
void * memcpy (void * destination, const void * source, size_t num);
```

- копирование строки. Позволяет копировать не только 1 элемент, а целую последовательность данных.

Если работать с числом байт для копирования, то можно превратить memcpy в splice.

△ memmove(void *des, void *source, size_t num) – копирует блок памяти из source в des. С разницей, что области могут пересекаться. Во время копирования используется промежуточный буфер, который предотвращает перекрытие областей.

△ strncpy(char* dest, char* source, size_t num) – копирует только num первых букв строки. 0 в конце не добавляется. – помни, что там передается указатель. Так если нужно начать с N символа, то выражение должно иметь вид &(A[n]).

△ strncat(char* dest, char* source, size_t num) – добавляет к концу строки dest строку source по длине num.

△ memchr(void *ptr, int value, size_t num); - проводит поиск среди первых num байтов участка памяти на который ссылается ptr на элемент value. Возвращает указатель на найденный элемент, либо NULL. Вызывается так:

```
ptr = (char*) memchr(str, 'o', 4000);
```

△ strchr(char *str, int character) – возвращает указатель на первое совпадение в строке по символу.

ВСЕ ГЛАСНЫЕ: [aeiouys](#)

△ strspn(char* string, char* etal) – возвращает длину участка, который состоит только из элементов массива etal

△ strstr(char* str, char* niece) – возвращает указатель на первое вхождение строки niece в строку str.

△ strtok(char* str, char* delim) – разделяет строку на “токены” и при этом возвращает указатель. При повторном вызове с аргументом NULL совершает шаг вперед до следующего слова и так пока не будет возвращен NULL.
- аналог split.

△ memset(void *ptr, int value, size_t num) – заполняет

△ int atoi(char* str) – переводит строку в число.

△ setlocale(int category, char* locale); -

RAND_MAX – возвращает MAX случайное значение.

Вычисление интеграла с помощью случайных чисел.

Элементы, что ниже функции – к одной группе, больше – другая группа. Отношение – площадь интеграла по границам области.

★ Некоторые дроби невозможно представить в 2ой виде, так как они становятся бесконечными. Бесконечные дроби не получается уместить в 24 бита для М. Тогда появляется ошибка.

Строковый литерал

char *str = “String literal” – строковый литерал, объявлен через *, а не через [].
printf(“%s”, str)

* его нельзя модифицировать.

Действительно str[0] = ‘s’ – выводит код в ошибку, которая даже не отображается. Это происходит, потому что у нас нет прав на модификацию объекта. Для предотвращения подобного необходимо указывать явно, что объект не модифицируемый.

Const char *str = “String Literal”;

△ assert(*str != NULL) – проверка входных данных на соответствие. Если там, что то не то, вы выбивается ошибка.

Структуры

В структуре можно хранить указатель этого типа:

```
struct node {  
    void* value;  
    struct node *next;
```

}.. для того что бы ссылаться на другие структуры, которые могут хранится в ней..

Инициализация: struct gasket obj = {10.2, 6, 8};

Можно определять структуры через идентификатор:

Классический вариант работы со структурой предполагает создание типа данных:
typedef struct Model {... } Model. Далее она используется в malloc, приведениях.

* Указатели на вложенные структуры возможны только тогда, когда структура определена.

* при копировании указателей на новые места необходимо копировать содержимое, а не просто указатели.

Перечисляемый тип (enum)

- Задающий набор всех возможных целочисленных значений переменной этого типа enum Gender {a, b, c}. Можно создавать типы)) но “константы” #define мне нравятся больше.

Классы памяти

register int x = 30 – переменная располагается в регистре, а не в оперативной памяти. Компилятор может её сделать таковой, если позволяют условия....

static long long prevAns = 1 – хранятся в data и bss сегменте. Их время жизни совпадает с временем жизни приложения. ★★ – собственный state функции. Это жестко. Можно делать кэши)))

* Способ реализации функции malloc не определен в документации языка. Известно, что выделенный участок памяти начинается с метаданных.

Продвинутое использование MALLOC

1. Создается структура метаданных с полями для хранения инфы. Сразу же преобразуется в тип данных. *Можно хранить кол-во элементов.

2. Создается функция malloc, которая принимает размер области памяти.. Создает указатель на эту область, но при этом учитывается размер структуры метаданных.

Void *ptr = malloc(size + sizeof(METADATA)); Проверяется факт выделения памяти. Далее указатель кастуется до указателя на структуру.. и в эту структуру через -> записывается инфа.

((METADATA*) ptr) -> size = size; - приводится к указателю на метаданные. ...

После записи происходит сдвиг вправо на размер указателя на метаданные. Что бы отдать как бы 2ую часть пространства памяти под использование.

GET_SIZE

Отдается указатель. Он там переводится до нулевой позиции. Это вот как происходит: void *zeroPos = (char*) ptr – sizeof(METADATA); То есть в рабочем состоянии указатель сдвинут на размер структуры меты и теперь необходимо сдвинуться назад, что бы получить доступ к структ.

Далее (условно char) указ преобразуется в указатель до (METADATA*); Далее можно ходить по параметрам структуры.

То есть чтобы получить доступ к структуре METADATA.

Память в программе

СТЕК – “последним пришёл, первым вышел (LIFO)”.

при работе с ним, есть вероятность получить ошибку “Stack overflow”, так как размер стека строго ограничен.

$\Delta_alloca(n)$ – выделение “быстрой” памяти на стеке. Легко переполняется и не возвращает ошибку.

Быстрое выделение памяти под динам. Массив

Суть в том, чтобы создать массив указателей, а затем первому элементу массива отдать размер памяти, который соответствует всем внутренним массивам. После этого вручную определить принадлежность области памяти индексу массива по верхнему уровню.

Функция с множеством аргументов

Передается число аргументов и аргументы). Далее работа с аргументами происходит в цикле, где происходит работа с указателем на рабочий аргумента. Он перебирается через плюсы.

★ При прямом обходе указателя через ++, нужно прибавлять 1. Даже если это float.. Они не показали, как работать с аргументами типа float.

Calloc работает стабильней???, чем Malloc?

Нахуй malloc