***Билеты по программированию 1 семестр 2020-2021 гг.***

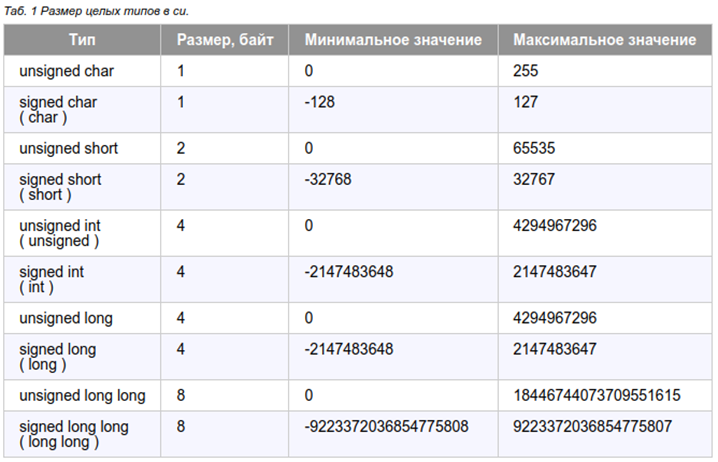
**Билет 1. Встроенные типы данных, представление в памяти.**

**Встроенные типы данных:**

Символьные – char

Целочисленные – int, short [int], long [int], unsigned [int]

Вещественные (числа с плавающей точкой, т.е. те числа, которые с дробной частью) – float, double, long double



Для получения размерности того или иного типа данных используется функция *sizeof():*

int main() {

char c;

short s;

int i;

long l;

long long L;

printf("sizeof(char) = %d\n", sizeof(c));

printf("sizeof(short) = %d\n", sizeof(s));

printf("sizeof(int) = %d\n", sizeof(i));

printf("sizeof(long) = %d\n", sizeof(l));

printf("sizeof(long long) = %d\n", sizeof(L));

return 0;

}

Вывод: sizeof(char) = 1

sizeof(short) = 2

sizeof(int) = 4

sizeof(long) = 4

sizeof(long long) = 8

**Представление чисел в памяти**

* BIN, OCT, DEC, HEX (2, 8, 10, 16)
* Двоичная алгебра

**Also представление чисел с помощью прямого кода, обратного кода и дополнительного кода:**

**Прямой код:**

Выделяем бит под знак:

▫ 0 - положительные

▫ 1 - отрицательные

• 710 = 0000 01112

• -710 = 1000 01112

• Диапазон чисел: []

• Достоинства:

▫ простота

▫ количество положительных = количество отрицательных

• Недостатки

▫ Сложность арифметических операций

▫ Два нуля.

**Обратный код:**

* Инвертируем биты

• 710 = 0000 01112

• -710 = 1111 10002

• Диапазон чисел: : []

• Достоинства:

▫ Вычитание легко заменяется сложением

• Недостатки

▫ Два нуля.

▫ Необходимо добавлять единицу переноса из знакового разряда к результату сложения

**Дополнительный код:**

• К коду отрицательного числа добавляем 1

• 710 = 0000 01112

• -710 = 1111 10012

• Диапазон чисел: []

**Представление вещественных чисел:**

• Числа с плавающей точкой

• N = m x bе

▫ b - основание

▫ e - порядок

▫ m - мантиса

• Нормальная форма

▫ 0 |m| < 1

**Представление символьных констант:**

• Символьная константа - ‘x’

• Некоторые символьные константы начинаются с эскейп последовательности ‘\n’

▫ символ новой строки ‘\n’

▫ горизонтальная табуляция ‘\t’

▫ обратная слеш ‘\\’

▫ одиночная кавычка ' \‘

• Строковая константа “Hello \’ world\’\n”

В вычислительных машинах символы не могут храниться иначе, как в виде последовательностей бит (как и числа). Для передачи символа и его корректного отображения ему должна соответствовать уникальная последовательность нулей и единиц. Для этого были разработаны таблицы кодировок.

Количество символов, которые можно задать последовательностью бит длины n, задается простой формулой C(n)=. Таким образом, от нужного количества символов напрямую зависит количество используемой памяти.

**Билет 2. Встроенные типы данных. Приведения типов. Идентификаторы. Объявление. Иницилизация. Область видимости переменных**.

**Приведения типов:**

• *Неявные* преобразования:

▫ Если какой-либо из операндов принадлежит типу long double, то и

другой приводится к long double.

▫ В противном случае, если какой-либо из операндов принадлежит типу

double, то и другой приводится к double.

▫ В противном случае, если какой-либо из операндов принадлежит типу

float, то и другой приводится к float.

▫ В противном случае операнды типов char и short приводятся к int.

▫ И наконец, если один из операндов типа long, то и другой приводится к

long.

• *Явное* преобразование (тип) переменная

т. е.

int main() {

int number = 70;

char symbol = (char)number; //(при неявном не было бы (char))

printf("symbol = %c \n", symbol);

printf("symbol (int code) = %d \n", symbol);

return 0;

}

**Идентификаторы:**

● Конструкции и элементы программы нужно как-то называть

● *Идентификаторы* – это имена, используемые для обозначения переменных, типов, функций, меток

● Идентификаторы создаются при декларировании переменных, типов или функций

Правила:

1. Буквы латинского алфавита, цифры и “\_”

2. Первый символ - буква

3. Прописные и строчные различаются (т.е. Value, и value, и VALUE – отличаются)

4. Длина не больше 31(63) символа (начиная с C99)

5. Не могут совпадать с ключевыми словами

**Объявление (declaration):**

int a;

char b, c, d;

signed int I;

**Иницилизация (определение переменных (definition)):**

int a = 0;

double r = 1.23;

float b = 23.5f;

float c = 1.0e-3;

char ch = 'c';

long l = 23456789L;

int i = 0X1F;

long double ld = 1.2345l;

**Область видимости переменных:**

Я бы сказала, есть два вида – локальная и глобальная. Глобальная – переменная, «живущая» вне функций. Доступ к глобальным переменным возможен из любого места в файле.

Переменные, объявленные в функции (в том числе и в main), будут существовать только в этих местах.

Пример:

#include <stdio.h>

#include <stdlib.h>

int global = 700;

int func(int a, int b) {

int c = a + b + global;

return c;

}

int main() {

int a = 6;

int b = 7;

int d = func(a, b);

printf("%d", d);

return 0;

}

В данной программе доступ к глобальной переменной global может быть в любом месте программы. Переменные a и b будут существовать только в функции func. Взять эти же переменные в main’е без объявления – невозможно. Необходимо создать их: объявить и определить. Так же нет доступа к переменной с из func’а, так как эта переменная существует только в этой функции. Время существования == время выполнения функции.

**Билет 3. Расположение данных в памяти. Доступ к данным, адреса, указатели. Операции с указателями.**

**Расположение данных в памяти: пока хз, что туда писать**

**Указатель**

● Указатель (pointer) – переменная, диапазон значений которой состоит из адресов ячеек памяти и специального значения – нулевого адреса

● Указатель «указывает» хранящимся внутреннего адресом на ячейку памяти, к которой с его помощью можно обратиться

● Значение нулевого адреса используется только для обозначения того, что указатель в данный момент не указывает ни на какую ячейку памяти

Унарный оператор & выдает адрес объекта

a = &b;

Унарный оператор \* есть оператор косвенного доступа

*Связь массивов и указателей*

● Определим массив

int a[10];

● Определим указатель

int \*pa;

● Присвоим указатель адресу первого элемента массива

pa = &a[0];

● Получим значение первого элемента массива через указатель int x = \*pa;

● Получим указатель на следующий элемент массива \*(pa + 1)

● Получим указатель на произвольный элемент массива \*(pa + i) – это эквивалентно a[i]

● Компилятор преобразует ссылку на массив в указатель на начало массива, следовательно:

○ Имя массива является указательным выражением

○ Записи pa = &a[0] и pa = a эквивалентны

○ Записи a[i], \*(a + i), \*(pa + i) и pa[i] эквивалентны

○ Массив можно объявлять, как указатель, а потом пользоваться им, как массивом

**Операции с указателями**

1. *ПРИСВАИВАНИЕ*. Указателю можно присвоить адрес.

2. *ОПРЕДЕЛЕНИЕ ЗНАЧЕНИЯ*. Операция выдает значение, хранящееся в указанной ячейке.

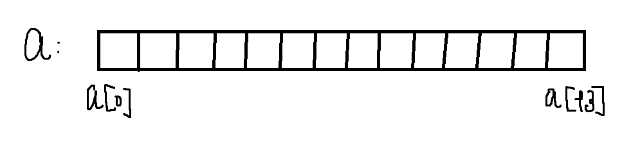
3. *ПОЛУЧЕНИЕ АДРЕСА УКАЗАТЕЛЯ*. Подобно любым переменным переменная типа указатель имеет адрес и значение. Операция & сообщает нам, где находится сам указатель.

4. *УВЕЛИЧЕНИЕ УКАЗАТЕЛЯ*. Мы можем выполнять это действие с помощью обычной операции сложения либо с помощью операции увеличения. Увеличивая указатель, мы перемещаем его на следующий элемент массива.

5. *РАЗНОСТЬ*. Можно находить разность двух указателей. Обычно это делается для указателей, ссылающихся на элементы одного и того же массива, чтобы определить, на каком расстоянии друг от друга находятся элементы. Помните, что результат имеет тот же тип, что и переменная, содержащая размер массива.

**Билет 4. Массивы. Объявление и инициализация. Устройство в памяти. Операции над массивами. Двумерные массивы. Арифметика указателей.**

**Программа для объяснения, что такое массив. Объявление и инициализация:**

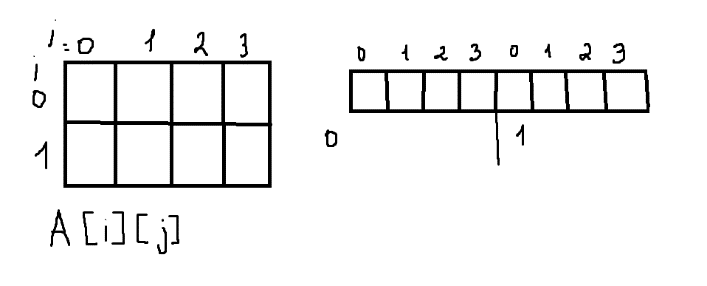
**Массив** — это непрерывный участок памяти, содержащий последовательность объектов одинакового типа, обозначаемый одним именем.

int main() {

int arr[10];

int arr2[] = { 1, 2, 3, 4, 5 };

int arr3[3] = { 1, 2, 3 };

 int arr4[2][3] = {

{1, 2, 3},

{4, 5, 6}

};

printf("%d\n", arr2[0]);

printf("%d\n", arr4[1][2]);

return 0;

}

***Операции над массивами***

###### 1)Заполнение массива элементами псевдослучайной последовательности (проблем не должно быть)

2)**Суммирование элементов массива (тем более проблем не должно быть)**

3)**Циклический сдвиг элементов массива**

int main() {

int a[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

int c = a[9];

for (int i = 9; i > 0; --i) {

a[i] = a[i - 1];

}

a[0] = c;

for (int i = 0; i < 10; ++i) {

if (i == 9) {

printf("%d.", a[i]);

break;

}

printf("%d, ", a[i]);

}

printf("\n");

return 0;

}

4)**Поиск в массиве**

Поиск максимума и минимума

Поиск числа К в массиве

Поиск числа К в упорядоченном массиве

5) **Сортировка массива** – на самом деле много вариаций сортировок. Так что не только пузырек. Можно реализовать любую сортировку из тех, что были на алгосах.

**Арифметика указателей**

int main() {

int a[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

int\* p;

p = a;

printf("%d \n", \*p);

p++;

printf("%d \n", \*p);

p += 6;

printf("%d \n", \*p);

return 0;

}

**Билет 5. Строки в С. Операции со строками стандартной библиотеки.***Строки*

● Массив символов

● Заключается в “”

● Escape character

● Null-terminated string

Строку можно выразить как с помощью массива, так и с помощью указателей. То есть:

char\* str;

char second\_str [] = "now";

*Операции со строками стандартной библиотеки:*

1. *Длина строки:*

int main() {

char\* str;

str = "Hello world!";

int count = 0;

while (\*str != '\0') {

str++;

count++;

}

printf("%d \n", count);

}

1. *Поиск символа: код, написанный у Димы, неплохой, но мне мой нравится больше.*

int main() {

char\* str;

str = "Hello world!";

char g;

int k = 0;

scanf("%c", &g);

int count = 0;

while (\*str != '\0') {

if (\*str == g) {

printf("the character in the %d place in the line\n", count + 1);

k++;

}

str++;

count++;

}

printf("%d \n", k);

}

1. *Сравнение строк:*

#include <stdio.h>

#include <stdlib.h>

int stringCompare(char\* first, char\* second) {

int i = 0;

while (first[i] != '\0' && second[i] != '\0') {

if (first[i] != second[i])

return first[i] < second[i] ? -1 : 1;

i++;

}

return first[i] == second[i] ? 0 : first[i] < second[i] ? -1 : 1;

}

void test(char\* first, char\* second) {

int r = stringCompare(first, second);

printf("[%s] %c [%s] \n", first, r == 0 ? '=' : r > 0 ? '>' : '<', second);

}

int main() {

test("", "");

test("ab", "a");

test("a", "ab");

test("abc", "");

test("", "abc");

test("abc", "Abc");

test("Abc", "abc");

test("abc", "abc");

return 0;

}

Я написала в два столбца, чтобы было меньше пустого пространства.

**Билет 6. Операции. Типы операций, приоритеты. Битовые и логические операции.**

**Операторы (как дополнительная информация – заканчивается ; , составной оператор выглядит следующим образом: {оператор1; оператор2; оператор3;}**

● Арифметические (+, -, \*, /, %)

● Сравнение (>, >=, <, <=, ==, !=)

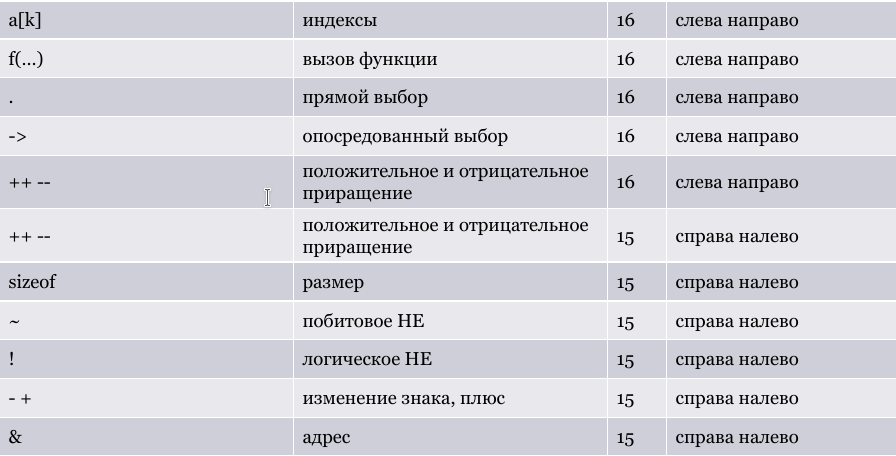
● Логические (&&, ||)

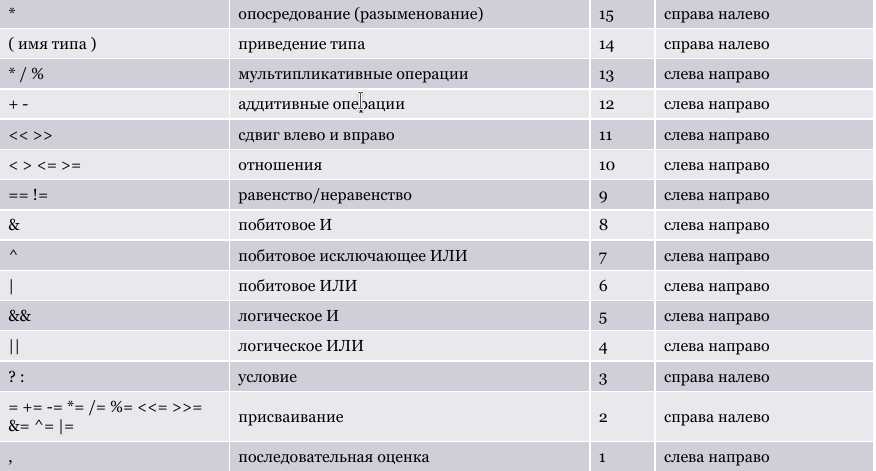
● Инкремента и Декремента (++, --)

● Побитовые (&, |, ^, <<, >>, ~)

● Присваивание (=, +=, \*=, )

● Условный (?:)

****

****

# **Битовые операции**

& - конъюнкция (логическое И, умножение)

| - дизъюнкция (логическое ИЛИ, сложение)

~ - побитовое отрицание (НЕ)

^ - XOR (Исключающее ИЛИ)

## Операции побитового сдвига

<< - побитовый сдвиг влево

**>> -** побитовый сдвиг вправо

**Логические операции**

! - логическое НЕ

&& - логическое И

|| - логическое ИЛИ

**Билет 7. Операторы ветвления. Тут и так понятно. Помним про && и ||, break, continue.**

#include <stdio.h>

#include <stdlib.h>

int main() {

//if-else

int a = 6;

int b = 7;

if (a > b)

printf("%d\n", a + b);

else

printf("%d\n", a \* b);

//else-if

if (a == b)

printf("%d\n", a + b);

else if (a > b)

printf("%d\n", a / b);

else

printf("%d\n", a \* b);

//switch. если у некоторых условий одинаковое выполнение той или иной функции, то:

//case 0:

//case 1:

//printf("%d\n", 0);

switch (a) {

case 0:

printf("%d\n", 0);

break;

case 1:

printf("%d\n", 1);

break;

case 2:

printf("%d\n", 2);

break;

default:

printf("%d\n", -1);

}

return 0;

}

Может быть такая запись: (a > b) ? a + b: a \* b;

*Помним про стиль кода*. Все операции необходимо заключать перед и после в пробелы. Выражение 1 в такой записи рекомендуется заключать в скобки. Ну тут ясно, мне еще про стиль не хватало писать.

Стоит ли писать о циклах? Не видела еще в билетах, но напомню:

//do-while

int g;

do

g = a + b;

while ((a / 10) > 0);

printf("%d\n", g);

//for

for (int n = 3; n > 0; n--)

printf("%d\n", n);

Немного о switch:

• Константы – целые

• Вычисления начинаются с первой совпавшей с константой в

ветке и выражения

• Все константы должны быть разные

• Если совпадения не нашлось, то выполняется default

• Break вызывает выход из switch

• Сквозное выполнение

**Билет 8. Функции. Объявления, реализация. Механизмы передачи и возврата данных. Механизм вызова функции.**

**Общая картина функции:**

тип имя(параметр1, параметр2…) {

объявления и инструкции

}

**Функция** — это самостоятельная единица программы, которая спроектирована для реализации конкретной подзадачи.

Функция является подпрограммой, которая может содержаться в основной программе, а может быть создана отдельно (в библиотеке). Каждая функция выполняет в программе определенные действия.

Сигнатура функции определяет правила использования функции. Обычно сигнатура представляет собой описание функции, включающее имя функции, перечень формальных параметров с их типами и тип возвращаемого значения.

Семантика функции определяет способ реализации функции. Обычно представляет собой тело функции.

Определение функции

Каждая функция в языке Си должна быть определена, то есть должны быть указаны:

• тип возвращаемого значения;

• имя функции;

• информация о формальных аргументах;

• тело функции.

В языке Си нельзя определять одну функцию внутри другой.

int max(int a, int b); //Объявление, тип до вызова если функцию не написал

int max(int a, int b) { // definition (определение)

*Передача данных* в функции вызывается в мэйне. Мы пишем те данные, которые необходимы для выполнения. Эти данные будут присваиваться определенным переменным для каждой функции. *Возврат данных* осуществляется с помощью оператора return.

Ну *вызов* понятен. Так же можно вызвать в мэйне или в другой функции. Вызывать в зависимости от возвращаемых данных (чаще всего присваивается какой-то переменной).

**Билет 9. Пользовательские типы данных.**

***Перечисления (enum)***

При написании программ часто возникает потребность определить несколько именованных констант, для которых требуется, чтобы все они имели различные значения (при этом конкретные значения могут быть не важны). Для этого удобно воспользоваться перечисляемым типом данных, все возможные значения которого задаются списком целочисленных констант. Формат:

enum [ имятипа ] { список констант };

Имя типа задается в том случае, если в программе требуется определять переменные этого типа. Компилятор обеспечивает, чтобы эти переменные принимали значения только из списка констант. Константы должны быть целочисленными и могут инициализироваться обычным образом. При отсутствии инициализатора первая константа обнуляется, а каждой следующей присваивается на единицу большее значение, чем предыдущей:

enum {two = 2, three, four, ten = 10, eleven, fifty = ten + 40};

Константам three и four присваиваются значения 3 и 4, константе eleven — 11.

Имена перечисляемых констант должны быть уникальными, а значения могут совпадать. Преимущество применения перечисления перед описанием именованных констант и директивой #define состоит в том, что связанные константы нагляднее; кроме того, компилятор при инициализации констант может выполнять проверку типов.

При выполнении арифметических операций перечисления преобразуются в целые. Поскольку перечисления являются типами, определяемыми пользователем, для них можно вводить собственные операции.

***Структуры (struct)***

В отличие от массива, все элементы которого однотипны, структура может содержать элементы разных типов. В языке C++ структура является видом класса и обладает всеми его свойствами, но во многих случаях достаточно использовать структуры так, как они определены в языке С:

struct [ имя\_типа ] {

тип\_1 элемент\_1:

тип\_2 элемент\_2;

тип\_n элемент\_n;

} [ список\_описателей ];

Элементы структуры называются полями структуры и могут иметь любой тип, кроме типа этой же структуры, но могут быть указателями на него. Если отсутствует имя типа, должен быть указан список описателей переменных, указателей или массивов. В этом случае описание структуры служит определением элементов этого списка:

// Определение массива структур и указателя на структуру:

struct {

char f1o[30];

int date, code;

double salary;

}staff[100], \*ps;

Если список отсутствует, описание структуры определяет новый тип, имя которого можно использовать в дальнейшем наряду со стандартными типами, например:

struct Worker{ // описание нового типа Worker

char f1o[30];

int date, code;

double salary;

}; // описание заканчивается точкой с запятой

// определение массива типа Worker и указателя на тип Worker:

Worker staff[100], \*ps;[/ccie\_cpp]

Имя структуры можно использовать сразу после его объявления (определение можно дать позднее) в тех случаях, когда компилятору не требуется знать размер структуры, например:

struct List;. // объявление структуры List

struct Link{

List \*p; // указатель на структуру List

Link \*prev, \*succ; // указатели на структуру Link

};

struct List { / \* определение структуры List \* / };

Это позволяет создавать связные списки структур.

Для инициализации структуры значения ее элементов перечисляют в фигурных скобках в порядке их описания:

struct{

char fio[30];

int date, code;

double salary;

}worker = {"Страусенке". 31. 215. 3400.55};

При инициализации массивов структур следует заключать в фигурные скобки каждый элемент массива (учитывая, что многомерный массив — это массив массивов):

struct complex{

float real, im;

} compl [2][3] = {

{{1. 1}. {1. 1}. {1. 1}}. // строка 1. TO есть массив compl[0]

{{2. 2}. {2. 2}. {2. 2}} // строка 2. то есть массив compl[1]

};

Для переменных одного и того же структурного типа определена операция присваивания, при этом происходит поэлементное копирование. Структуру можно передавать в функцию и возвращать в качестве значения функции. Другие операции со структурами могут быть определены пользователем. Размер структуры не обязательно равен сумме размеров ее

элементов, поскольку они могут быть выровнены по границам слова.

Доступ к полям структуры выполняется с помощью операций выбора . (точка) при обращении к полю через имя структуры и -> при обращении через указатель, например:

Worker worker, staff[100], \*ps;

worker.fio = "Страусенке";

staff[8].code = 215;

ps->salary = 0.12;

Если элементом структуры является другая структура, то доступ к ее элементам выполняется через две операции выбора:

struct А {int а; double х;};

struct В {А а; double х;} х[2];

х[0].а.а = 1;

х[1].х = 0.1;

Как видно из примера, поля разных структур могут иметь одинаковые имена, поскольку у них разная область видимости. Более того, можно объявлять в одной области видимости структуру и другой объект (например, переменную или массив) с одинаковыми именами, если при определении структурной переменной использовать слово struct, но не советую это делать — запутать компилятор труднее, чем себя.

**Битовые поля**

Битовые поля — это особый вид полей структуры. Они используются для плотной упаковки данных, например, флажков типа «да/нет». Минимальная адресуемая ячейка памяти — 1 байт, а для хранения флажка достаточно одного бита. При описании битового поля после имени через двоеточие указывается длина поля в битах (целая положительная константа):

struct Options{

bool centerX:1;

bool centerY:1;

unsigned int shadow:2;

unsigned int palette:4;

};

Битовые поля могут быть любого целого типа. Имя поля может отсутствовать, такие поля служат для выравнивания на аппаратную границу. Доступ к полю осуществляется обычным способом — по имени. Адрес поля получить нельзя, однако в остальном битовые поля можно использовать точно так же, как обычные поля структуры. Следует учитывать, что операции с отдельными битами реализуются гораздо менее эффективно, чем с байтами и словами, так как компилятор должен генерировать специальные коды, и экономия памяти под переменные оборачивается увеличением объема кода программы. Размещение битовых полей в памяти зависит от компилятора и аппаратуры.

Объединения (union)

Объединение (union) представляет собой частный случай структуры, все поля которой располагаются по одному и тому же адресу. Формат описания такой же, как у

**Билет 10. Препроцессор языка С. Макросы. Условная компиляция. Include guard. Комментарии.**

Препроцессор С/С++ (англ. pre processor, предобработчик) — программа, подготавливающая код программы на языке C/C++ к компиляции.

*Препроцессором выполняются следующие действия:*

* замена соответствующих диграфов и триграфов на эквивалентные символы «#» и «»;
* удаление экранированных символов перевода строки;
* замена строчных и блочных комментариев пустыми строками (с удалением окружающих пробелов и символов табуляции);
* вставка (включение) содержимого произвольного файла (#include);
* макроподстановки (#define);
* условная компиляция (#if, #ifdef, #elif, #else, #endif);
* вывод сообщений (#warning, #error).

*Условная компиляция позволяет выбрать код для компиляции в зависимости от:*

* модели процессора (платформы);
* разрядности адресов;
* размерности типов;
* наличия/отсутствия поддержки расширений языка;
* наличия/отсутствия библиотек и/или функций;
* особенностей поведения конкретных функций;
* и другого.

*Этапы работы препроцессора:*

* лексический анализ кода C/C++ (синтаксический анализ не выполняется);
* обработка директив;
* выполнение подстановок:
* диграфов и триграфов;
* комментариев;
* директив;
* лексем, заданных директивами.

Язык препроцессора C/C++ не является полным по Тьюрингу хотя бы потому, что с помощью директив невозможно заставить препроцессор зависнуть. См. рекурсивная функция (теория вычислимости).

Перед тем как программа будет скомпилирована (или не будет, если найдены ошибки), текст программы обрабатывается препроцессором. Препроцессор позволяет изменять текст программы, используя специальные директивы.

Директива #define определяет новый макрос. Макрос, или макроподстановка, будет заменена в коде программы своим телом.

#pragma once

**Препроцессор**

● Язык препроцессора – это специальный язык программирования,

встроенный в C. ∙

● Препроцессор работает с кодом на C как с текстом. ∙

● Команды языка препроцессор называют директивами, все директивы

начинаются со знака #.

● Директива #include позволяет подключать заголовочные файлы к

файлам кода.

● Препроцессор заменяет директиву #include "bar.h" на содержимое файла

bar.h.

1. #define и #unded (\_\_FILE\_\_, \_\_LINE\_\_, ...)

2. # and ## operators

3. Условное включение( #if #ifdef #ifndef #elif #else #endif)

4. #include

a. #include <filename>

b. #include "filename"

5. #pragma once

6. #error

**Макросы**

● Макросами в C называют инструкции препроцессора. ∙

● Препроцессор C является самостоятельным языком, работающим с произвольными строками. ∙

● Макросы можно использовать для определения функций: Препроцессор “не знает” про синтаксис C++.

## Условная компиляция

Директивы #if или #ifdef/#ifndef вместе с директивами #elif, #else и #endif управляют компиляцией частей исходного файла. Если указанное выражение после #if имеет ненулевое значение, в записи преобразования сохраняется группа строк, следующая сразу за директивой #if.

Отличие директив  #ifdef/#ifndef заключается в том, что константное выражение может быть задано только с помощь #define.  
  
У каждой директивы #if в исходном файле должна быть соответствующая закрывающая директива #endif. Между директивами #if и #endif может располагаться любое количество директив #elif, однако допускается не более одной директивы #else. Директива #else, если присутствует, должна быть последней перед директивой #endif.

Информации много, можно так же почитать у Димы, но заучивать не стоит, проще понять.

**Билет 11. Работа с файлами. Режимы открытия файлов. Общие сведения о потоках ввода-вывода в языке С.**

**Файл** – именованная область внешней памяти, выделенная для хранения массива данных. Данные, содержащиеся в файлах, имеют самый разнообразный характер: программы на алгоритмическом или машинном языке; исходные данные для работы программ или результаты выполнения программ; произвольные тексты; графические изображения и т. п.

Для программиста открытый файл представляется как последовательность считываемых или записываемых данных. При открытии файла с ним связывается поток ввода-вывода. Выводимая информация записывается в поток, вводимая информация считывается из потока.

Когда поток открывается для ввода-вывода, он связывается со стандартной структурой типа FILE, которая определена в stdio.h. Структура FILE содержит необходимую информацию о файле.

Открытие файла осуществляется с помощью функции fopen(), которая возвращает указатель на структуру типа FILE, который можно использовать для последующих операций с файлом.

FILE \*fopen(name, type);

name – имя открываемого файла (включая путь),

type — указатель на строку символов, определяющих способ доступа к файлу:

"r" — открыть файл для чтения (файл должен существовать);

"w" — открыть пустой файл для записи; если файл существует, то его содержимое теряется;

"a" — открыть файл для записи в конец (для добавления); файл создается, если он не существует;

"r+" — открыть файл для чтения и записи (файл должен существовать);

"w+" — открыть пустой файл для чтения и записи; если файл существует, то его содержимое теряется;

"a+" — открыть файл для чтения и дополнения, если файл не существует, то он создаётся.

Возвращаемое значение — указатель на открытый поток. Если обнаружена ошибка, то возвращается значение NULL.

Функция fclose() закрывает поток или потоки, связанные с открытыми при помощи функции fopen() файлами. Закрываемый поток определяется аргументом функции fclose().

Возвращаемое значение: значение 0, если поток успешно закрыт; константа EOF, если произошла ошибка.

Функции fscanf() и fprintf() аналогичны функциям scanf() и printf(), но работают с файлами данных, и имеют первый аргумент — указатель на файл.

fscanf(поток, "ФорматВвода", аргументы);

fprintf(поток, "ФорматВывода", аргументы);

Функции fgets() и fputs() предназначены для ввода-вывода строк, они являются аналогами функций gets() и puts() для работы с файлами.

fgets(УказательНаСтроку, КоличествоСимволов, поток);

Символы читаются из потока до тех пор, пока не будет прочитан символ новой строки ‘\n’, который включается в строку, или пока не наступит конец потока EOF или не будет прочитано максимальное количество символов. Результат помещается в указатель на строку и заканчивается нуль- символом ‘\0’. Функция возвращает адрес строки.

**Билет 12. Сборка программ. Препроцессор, компилятор, компоновщик. Сборка проекта, состоящего из нескольких файлов. Макросы.**

Компиляция исходных текстов на Си в исполняемый файл происходит в три этапа, которых на самом деле 8

**Препроцессинг**

Эту операцию осуществляет текстовый препроцессор. Исходный текст частично обрабатывается — производятся:

* Замена комментариев пустыми строками
* Текстовое включение файлов — #include
* Макроподстановки — #define
* Обработка директив условной компиляции — #if, #ifdef, #elif, #else, #endif

**Компиляция**

Процесс компиляции состоит из следующих этапов:

* Лексический анализ. Последовательность символов исходного файла преобразуется в последовательность лексем.
* Синтаксический анализ. Последовательность лексем преобразуется в дерево разбора.
* Семантический анализ. Дерево разбора обрабатывается с целью установления его семантики (смысла) — например, привязка идентификаторов к их декларациям, типам, проверка совместимости, определение типов выражений и т. д.
* Оптимизация. Выполняется удаление излишних конструкций и упрощение кода с сохранением его смысла.
* Генерация кода. Из промежуточного представления порождается объектный код.

Результатом компиляции является объектный код.

*Объектный код* — это программа на языке машинных кодов с частичным сохранением символьной информации, необходимой в процессе сборки.

При отладочной сборке возможно сохранение большого количества символьной информации (идентификаторов переменных, функций, а также типов).

**Компоновка**

Также называется связывание, сборка или линковка.

Это последний этап процесса получения исполняемого файла, состоящий из связывания воедино всех объектных файлов проекта.

При этом возможны ошибки связывания. Если, допустим, функция была объявлена, но не определена, ошибка обнаружится только на этом этапе.

**Компиляция нескольких файлов:**

qwe.c

#include <stdio.h>

#include "qwe1.h"

int main() {

int x = 10;

int y = 5;

int z;

z = sum\_2number(x, y);

printf("%d \n", z);

return 0;

}

qwe1.c

int sum\_2number(int a, int b) {

int sum = 0;

sum = a + b;

return sum;

}

qwe1.h

#ifndef SUM\_2NUMBER

#define SUM\_2NUMBER

int sum\_2number(int, int);

#endif

**Еще раз о макросах можно прочесть в билете 10.**

#ifdef \_\_DEBUG\_\_

#define error\_log(format, ...) printf("[ERROR] (%s:%d) " format "\n", \_\_FILE\_\_,

\_\_LINE\_\_, ##\_\_VA\_ARGS\_\_)

#else

#define error\_log(format, ...)

#endif

int main() {

error\_log("fatal errro no: %d", 1);

return 0;

}

**2.**

#define MAX(x, y) x > y ? x : y

int main() {

printf("%d\n", MAX(10, 20));

return 0;

}

**3.**

#define log(type, format, ...) printf("[" type "] (%s:%d) " format "\n", \_\_FILE\_\_,

\_\_LINE\_\_, ##\_\_VA\_ARGS\_\_)

#define log\_error(...) log("ERORR", ##\_\_VA\_ARGS\_\_ )

#define log\_info(...) log("INFO", ##\_\_VA\_ARGS\_\_ )

**Билет 13. Запуск программ. Процессы и потоки. Виды памяти. Стек вызова функций.**

Способ хранения объекта в языке Си определяет его время жизни — часть времени выполнения программы, во время которого объект существует или для него зарезервировано место. Объект имеет постоянный адрес и сохраняет своё последнее значение. Запрещается обращаться к объекту, который перестал существовать, при этом, если при работе с объектом использовался указатель, его значение остаётся неопределённым.

В традиционных языках программирования, таких как *Си*, существуют три вида памяти: ***статическая, стековая и******динамическая (куча).***

1. Статический объект можно инициализировать явно, либо использовать умалчиваемую инициализацию. Два вида: *глобальная и статическая переменная*. **Статические переменные** - это переменные, в описании которых присутствует слово static. Как правило, статические переменные описываются **вне функций**. Такие статические переменные во всем подобны глобальным, с одним исключением: область видимости статической переменной ограничена одним файлом, внутри которого она определена, - и, более того, ее можно использовать только после ее описания, т.е. ниже по тексту. По этой причине описания статических переменных обычно выносятся в начало файла. В отличие от глобальных переменных, статические переменные **никогда** не описываются в h-файлах (модификаторы extern и static конфликтуют между собой). Совет: используйте статические переменные, если нужно, чтобы они были доступны только для функций, описанных внутри **одного и того же файла**. По возможности не применяйте в таких ситуациях глобальные переменные, это позволит избежать конфликтов имен при реализации больших проектов, состоящих из сотен файлов.
2. Стековая память. Локальные, или стековые, переменные - это переменные, описанные **внутри функции**. Память для таких переменных выделяется в аппаратном стеке. Память выделяется в момент входа в функцию или блок и освобождается в момент выхода из функции или блока. При этом захват и освобождение памяти происходят практически мгновенно, т.к. компьютер только изменяет регистр, содержащий адрес вершины стека.
3. При работе с динамической памятью возможны утечки памяти и ошибки двойного освобождения блока. В отличии от стека позволяет создавать динамические структуры большого размера. Управление жизнью объектов в куче “ручное”. Программа может захватывать участки динамической памяти нужного размера. После использования ранее захваченный участок динамической памяти следует освободить.

**Стек вызова функций** отслеживает все активные функции (те, которые были вызваны, но еще не завершены) от начала программы и до текущей точки выполнения, и обрабатывает выделение всех параметров функции и локальных переменных.

Функция malloc выделяет size байтов памяти и возвращает указатель на неё. Если память выделить не удалось, то функция возвращает NULL. Так как malloc возвращает указатель типа void, то его необходимо явно приводить к нужному нам типу. Например, создадим указатель, после этого выделим память размером в 100 байт.

Функция calloc выделяет n объектов размером m и заполняет их нулями. Обычно она используется для выделения памяти под массивы.

realloc (re-allocation). Она позволяет изменить размер ранее выделенной памяти и получает в качестве аргументов старый указатель и новый размер памяти в байтах:

*Не знаю, насколько это нужно, но:* Стек так важен, потому что он отслеживает функции, выполняемые в программе, а функции, в свою очередь, являются строительными блоками программного обеспечения. На самом деле внутренняя работа программ обычно очень проста. Он состоит в основном из функций, толкающих данные в стек и выскакивающих из него при вызове друг друга, а также выделяющих память в куче для данных, которые должны сохраняться при вызовах функций. Это верно как для низкоуровневого программного обеспечения C, так и для языков на базе виртуальных машин, таких как JavaScript и C#. Твердое понимание этой реальности бесценно для отладки, настройки производительности и вообще для понимания того, что, черт возьми, происходит

Сразу же на сцену ворвались три регистра процессора. Указатель стека, esp, указывает на верхнюю часть стека. Верхняя часть всегда занята последним предметом, который был помещен в стопку, но еще не был снят точно так же, как в реальной стопке тарелок или 100-долларовых банкнот.

Адрес, хранящийся в esp, постоянно меняется по мере того, как элементы стека выталкиваются и выскакивают, так что он всегда указывает на последний элемент. Многие инструкции процессора автоматически обновляют esp как побочный эффект, и использовать стек без этого регистра нецелесообразно.

В архитектуре Intel, как и в большинстве других, стек растет в сторону более низких адресов памяти. Таким образом, “верх” - это самый низкий адрес памяти в стеке, содержащем текущие данные: local\_buffer в данном случае. Обратите внимание, что в стрелке от esp до local\_buffer нет ничего расплывчатого. Эта стрелка означает бизнес: она указывает конкретно на первый байт, занятый local\_buffer, потому что это точный адрес, хранящийся в esp

***Поток***

○ Поток данных в программировании — абстракция, используемая

для чтения или записи файлов, сокетов и т. п. в единой манере.

○ Поддержка потоков включена в большинство языков программирования

○ При запуске процесса ему предоставляются

предопределенные стандартные потоки

○ Стандартные потоки ввода-вывода — потоки процесса, имеющие

номер (дескриптор), зарезервированный для выполнения некоторых

«стандартных» функций:

1. Стандартный ввод (stdin)

2. Стандартный вывод (stdout)

3. stderr

**Билет 14. Стандартная библиотека языка С.**

**Стандартной библиотекой языка Си** (также известная как libc, crt) называется часть стандарта ANSI C, посвященная заголовочным файлам и библиотечным подпрограммам. Является описанием реализации общих операций, таких как обработка ввода-вывода и строк, в языке программирования Си. Стандартная библиотека языка Си — это описание программного интерфейса, а не настоящая библиотека, пригодная для использования в процессе компиляции.

По факту язык Си самостоятельно, без функций стандартной библиотеки, почти ничего не может. Даже вывести значение переменной на экран. Но благодаря дополнительным модулям (стандартным заголовочным файлам) возможности языка могут быть существенно расширены. А если какие-то возможности не нужны, то они не будут включены в итоговую версию программы.

Модульный подход получил широкое распространение в программировании. Основные плюсы:

модули между собой независимы и каждый из них может разрабатываться/меняться отдельно от других модулей и всей программы в целом;

* возможность самостоятельно подключать необходимые модули;
* возможность расширения функционала за счёт добавления/написания нового модуля.

Модули могут быть реализованы, например, в виде отдельных заголовочных файлов или отдельных файлов с кодом. Каждый модуль должен решать какие-то свои задачи. Модуль можно представить себе как «чёрный ящик». У него есть две основных части: интерфейс взаимодействия с модулем (сколько и каких данных поступает в чёрный ящик и какие данные им возвращаются) и внутренняя реализация модуля (содержимое чёрного ящика).

Если рассматривать отдельную функцию языка Си как модуль, то интерфейсом является заголовок или прототип функции. Он полностью описывает, что необходимо функции для работы: какого типа данные нужны, в каком порядке их надо передавать, в каком количестве и т.д., а также какие данные функция возвращает. Тело функции при этом является реализацией модуля.

**Задачи библиотеки**

● основной набор математических функций

● обработка строк

● конвертация типов

● файловый и консольный ввод-вывод

Структура

● Стандартная библиотека ANSI Си состоит из 29 заголовочных файлов

● Каждый из которых можно подключать к программному проекту при

помощи одной директивы

● Каждый заголовочный файл содержит объявления одной или более

функций, определения типов данных и макросы.

**Список заголовочных файлов**

**Имя Описание**

<assert.h> Содержит макрос утверждений, используемый для обнаружения логических и

некоторых других типов ошибок в отлаживаемой версии программы

<complex.h> Набор функций для работы с комплексными числами.

<ctype.h> Содержит функции, используемые для классификации символов по их типам или для

конвертации между верхним и нижним регистрами независимо от

используемой кодировки

<errno.h> Для проверки кодов ошибок, возвращаемых библиотечными функциями.

<fenv.h> Для управления средой, использующей числа с плавающей запятой.

<float.h> Содержит заранее определенные константы, описывающие специфику реализации

свойств библиотеки для работы с числами с плавающей запятой

<inttypes.h> Для точной конвертации целых типов.

<iso646.h> Для программирования в кодировке ISO 646.

<limits.h> Содержит заранее заданные константы, определяющие специфику

реализации свойств целых типов, как, например, область допустимых

значений (\_MIN, \_MAX).

<locale.h> Для setlocale() и связанных констант. Используется для выбора

соответствующего языка.

<math.h> Для вычисления основных математических функций

<setjmp.h> Объявляет макросы setjmp и longjmp, используемые для переходов

<signal.h> Для управления различными исключительными условиями

<stdarg.h> Для доступа к различному числу аргументов, переданных функциям.

<stdbool.h> Для булевых типов данных.

<stdint.h> Для определения различных типов целых чисел.

<stddef.h> Для определения нескольких полезных типов и макросов.

<stdio.h> Реализует основные возможности ввода и вывода в языке Си.

<stdlib.h> Для выполнения множества операций, включая конвертацию, генерацию

псевдослучайных чисел, выделение памяти, контроль процессов,

окружения, сигналов, поиска и сортировки.

<string.h> Для работы с различными видами строк.

<tgmath.h> Для типовых математических функций.

<time.h> Для конвертации между различными форматами времени и даты.

<wchar.h> Unicode

<wctype.h> Unicode

Stdio.h.

● Работа с файлами

● Форматированный ввод-вывод

● Вывод ошибок