

Ответы на теоретическую часть к экзамену по “Программированию на Си” 2-3 семестр.



0. Ликбез по утилитам и прочее

- **make** — утилита, автоматизирующая процесс преобразования файлов из одной формы в другую. Чаще всего это компиляция исходного кода в объектные файлы и последующая компоновка в исполняемые файлы или библиотеки. Утилита использует специальные make-файлы, в которых указаны зависимости файлов друг от друга и правила для их удовлетворения. На основе информации о времени последнего изменения каждого файла make определяет и запускает необходимые программы.
`make [-f make-файл] [цель] ...`
- **ld** - объединяет несколько объектных файлов в один, размещает команды и данные, разрешает внешние ссылки и генерирует таблицу имен для символьной отладки. В простейшем случае задаются имена нескольких объектных программ, и редактор связей объединяет их в один объектный модуль, который может затем или выполняться, или использоваться в качестве исходного при последующих вызовах ld. Результат редактирования связей помещается в файл с именем a.out, который является

выполняемым, если во время работы не было зафиксировано ошибок. Если какой-либо исходный файл не является объектным, `ld` предполагает, что это либо текстовый файл с директивами для редактора связей, либо архивная библиотека.

- **GCC** - это свободно доступный оптимизирующий компилятор для языков C, C++, программа `gcc`, запускаемая из командной строки, представляет собой надстройку над группой компиляторов. В зависимости от расширений имен файлов, передаваемых в качестве параметров, и дополнительных опций, `gcc` запускает необходимые препроцессоры, компиляторы, линкеры.
- **Размер указателя** - 4 байта (8 для 64x систем)
- **POSIX** (Portable Operating System Interface) — набор стандартов, описывающих интерфейсы между операционной системой и прикладной программой (системный API), библиотеку языка C и набор приложений и их интерфейсов. Стандарт создан для обеспечения совместимости различных UNIX-подобных операционных систем и переносимости прикладных программ на уровне исходного кода, но может быть использован и для не-Unix систем

Задачи

- содействовать облегчению переноса кода прикладных программ на иные платформы;
- способствовать определению и унификации интерфейсов заранее при проектировании, а не в процессе их реализации;
- сохранять по возможности и учитывать все главные, созданные ранее и используемые прикладные программы;
- определять необходимый минимум интерфейсов прикладных программ, для ускорения создания, одобрения и утверждения документов;
- развивать стандарты в направлении обеспечения коммуникационных сетей, распределенной обработки данных и защиты информации;
- рекомендовать ограничение использования бинарного (объектного) кода для приложений в простых системах.

- **as** - the portable GNU assembler. The GNU Assembler, commonly known as `gas` or simply as, its executable name, is the assembler used by the GNU Project. It is the default back-end of GCC. It is used to assemble the GNU operating system and the Linux kernel, and various other software. It is a part of the GNU Binutils package.

1.2 The GNU Assembler

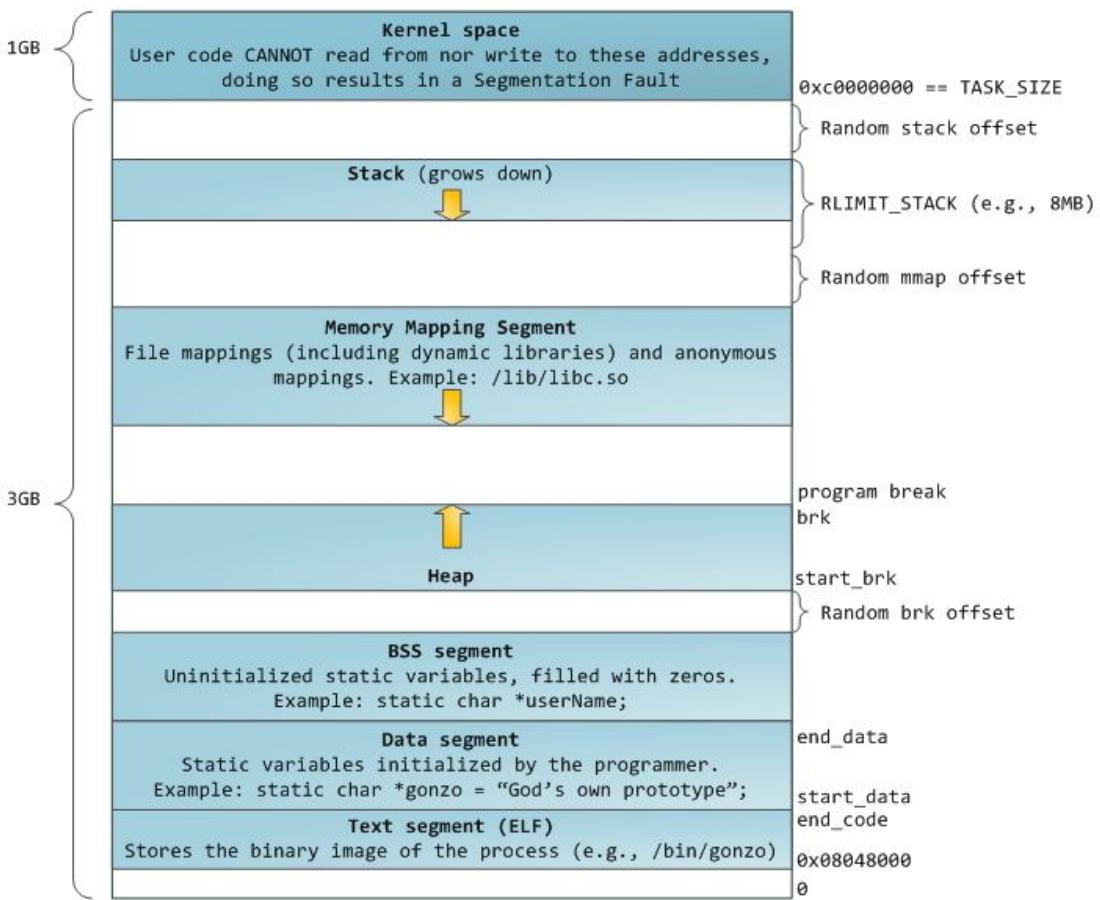
GNU `as` is really a family of assemblers. If you use (or have used) the GNU assembler on one architecture, you should find a fairly similar environment when you use it on another architecture. Each version has much in common with the others, including object file formats, most assembler directives (often called *pseudo-ops*) and assembler syntax.

`as` is primarily intended to assemble the output of the GNU C compiler `gcc` for use by the linker `ld`. Nevertheless, we've tried to make `as` assemble correctly everything that other assemblers for the same machine would assemble. Any exceptions are documented explicitly (see [Chapter 9 \[Machine Dependencies\]](#), page 79). This doesn't mean `as` always uses the same syntax as another assembler for the same architecture; for example, we know of several incompatible versions of 680x0 assembly language syntax.

Unlike older assemblers, `as` is designed to assemble a source program in one pass of the source file. This has a subtle impact on the `.org` directive (see [Section 7.82 \[.org\]](#), page 62).

```
as -o my-object-file.o mumble.s
```

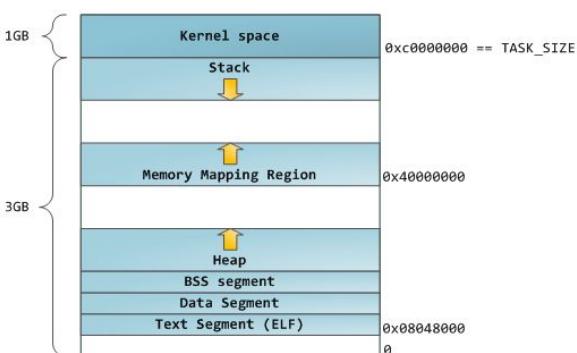
- **(Немного о памяти)** Каждый процесс в многозадачной ОС выполняется в собственной “песочнице”. Эта песочница представляет собой виртуальное адресное пространство, которое в 32-битном защищенном режиме всегда имеет размер равный 4 гигабайтам. Соответствие между виртуальным пространством и физической памятью описывается с помощью таблицы страниц (page table). Ядро создает и заполняет таблицы, а процессор обращается к ним при необходимости осуществить трансляцию адреса. Каждый процесс работает со своим набором таблиц. Есть один важный момент — концепция виртуальной адресации распространяется на все выполняемое ПО, включая и само ядро. По этой причине для него резервируется часть виртуального адресного пространства (т.н. kernel space).



- В верхней части user mode space расположена стековая область. Большинство языков программирования используют ее для хранения локальных переменных и аргументов, переданных в функцию. Вызов функции или метода приводит к помещению в стек т.н. стекового фрейма. Когда функция возвращает управление, стековый фрейм уничтожается. Стек устроен достаточно просто — данные обрабатываются в соответствии с принципом «последним пришёл — первым обслужен» (LIFO). По этой причине, для отслеживания содержания стека не нужно сложных управляющих структур — достаточно всего лишь указателя на верхушку стека. Добавление данных в стек и их удаление — быстрая и четко определенная операция. Более того, многократное

использование одних и тех же областей стекового сегмента приводит к тому, что они, как правило, находятся в кеше процессора, что еще более ускоряет доступ. Каждый тред в рамках процесса работает с собственным стеком.

- **Кстати о куче.** Она идет следующей в нашем описании адресного пространства процесса. Подобно стеку, куча используется для выделения памяти во время выполнения программы. В отличие от стека, память, выделенная в куче, сохранится после того, как функция, вызвавшая выделение этой памяти, завершится. Большинство языков предоставляют средства управления памятью в куче. Таким образом, ядро и среда выполнения языка совместно осуществляют динамическое выделение дополнительной памяти. В языке C, интерфейсом для работы с кучей является семейство функций malloc(), в то время как в языках с поддержкой garbage collection, вроде C#, основной интерфейс – это оператор new.
- **Наконец, мы добрались до сегментов**, расположенных в нижней части адресного пространства процесса: BSS, сегмент данных (data segment) и сегмент кода (text segment). BSS и data сегмент хранят данные, соответствующий static переменным в исходном коде на С. Разница в том, что в BSS хранятся данные, соответствующие неинициализированным переменным, чьи значения явно не указаны в исходном коде (в действительности, там хранятся объекты, при создании которых в декларации переменной либо явно указано нулевое значение, либо значение изначально не указано, и в линкуемых файлах нет таких же common символов, с ненулевым значением. – прим. перевод.). Для сегмента BSS используется анонимное отображение в память, т.е. никакой файл в этот сегмент не мэпируется. Если в исходном файле на С использовать int cntActiveUsers, то место под соответствующий объект будет выделено в BSS.
- Можно использовать утилиты nm и objdump для просмотра содержимого бинарных исполняемых образов: символов, их адресов, сегментов и т.д. Наконец, то, что описано в этом посте – это так называемая “гибкая” организация памяти процесса (flexible memory layout), которая вот уже несколько лет используется в Linux по умолчанию. Данная схема предполагает, что у нас определено значение константы RLIMIT_STACK. Когда это не так, Linux использует т.н. классическую организацию, которая изображена на рисунке:



PE¹⁰¹ a windows executable walkthrough

Ange Albertini
corkami.com

Dissected PE



Loading process



Notes

INT Import Name Table
Null-terminated list of pointers to Hint, Name structures
IAT Import Address Table
Null-terminated list of pointers
On file it is a copy of the INT
After loading it points to the Imported APIs
HINT
Index in the exports table of a DLL to be imported
Not required but provides a speed-up by reducing look-up

1. Язык программирования Си.

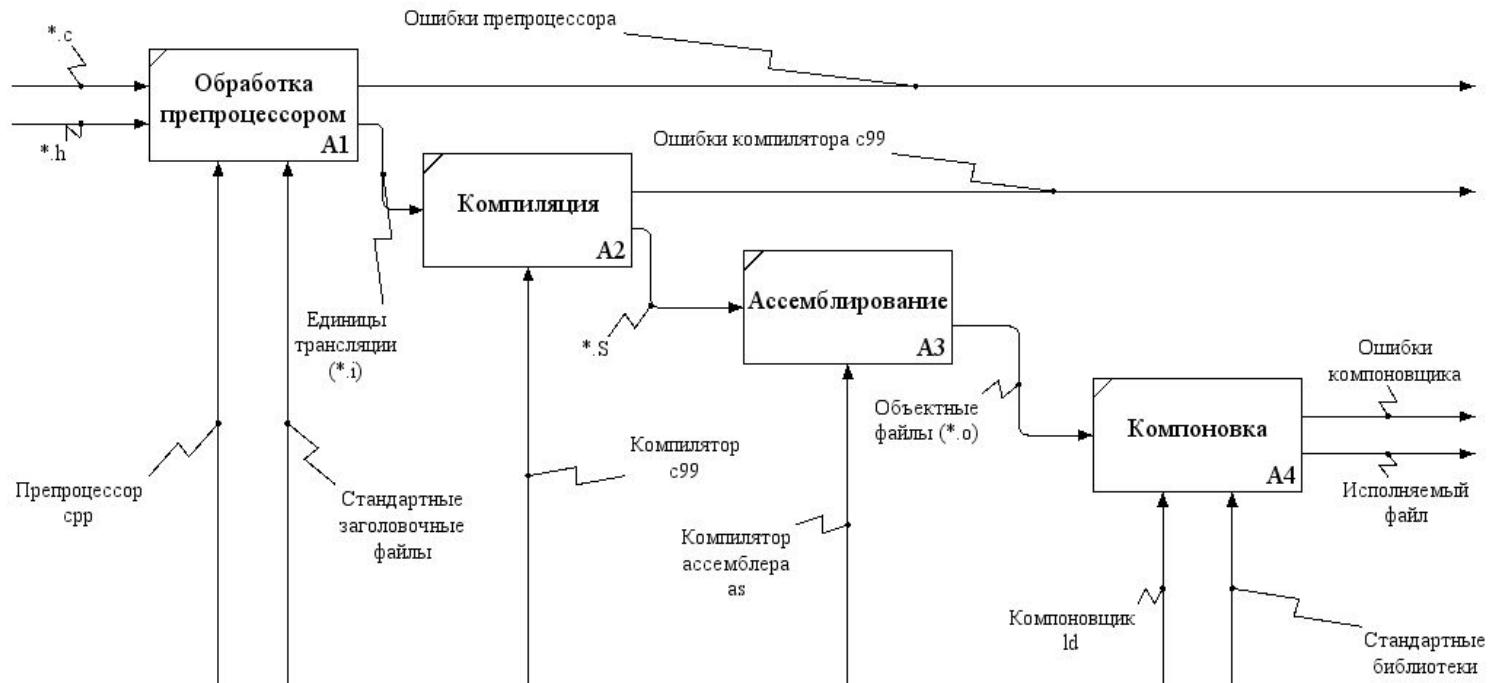
План ответа:

- История появления

- Разработан в 1969- 1973 годах Деннисом Ритчи и Кеном Томпсоном, первоначально был разработан для реализации операционной системы UNIX, но впоследствии был перенесён на множество других платформ.
- Конструкции близки к машинным инструкциям
- Синтаксис стал основой для C++, C#, Java и Objective-C.
- Расширения .c и .h
- ANSI C — стандарт языка С, опубликованный Американским национальным институтом стандартов (ANSI). Следование этому стандарту помогает создавать легко портируемые программы.(С89)
- Нововведения:

- `long long int`
- `snprintf`
- Встраиваемые функции (`inline`)
- **Особенности языка**
 - После публикации K&R C в язык было добавлено несколько возможностей, поддерживаемых компиляторами AT&T и некоторых других производителей:
 - функции, не возвращающие значение (с типом `void`), и указатели, не имеющие типа (с типом `void*`);
 - функции, возвращающие объединения и структуры;
 - спецификатор констант (`const`);
 - стандартная библиотека, реализующая большую часть функций, введенных различными производителями;
 - перечислимый тип (`enum`);
 - дробное число одинарной точности (`float`).
- **Использование**
 - Эти языки используются там, где нужно максимальное быстродействие, экономия памяти и "близость" к железу. Особенno это относится к Си. С++ уровнем чуть выше и у программ на нем требования к ресурсам чуть больше.
 - СУБД - Oracle*, MySQL*, SQL Server*, PostgreSQL

2. Этапы получения исполняемого файла из исходного кода. Опции компилятора и компоновщика.



План ответа

- Препроцессирование

- 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`
- `gcc -o main.i main.c`

- Компиляция

- Трансляция программы — преобразование программы, представленной на одном из языков программирования, в программу на другом языке. Транслятор обычно выполняет также диагностику ошибок, формирует словари идентификаторов, выдаёт для печати текст программы и т. д.
- `c99 -S -masm=intel main.i`

- Ассемблирование

- Ассемблирование в объектный файл
- `as -o main.o main.s`

- **Компоновка «стандартная» (POSIX) строка запуска компилятора**
 - `ld -o hello.exe hello.o`
- **ключи компилятора и компоновщика: -std, -Wall, -Werror, -pedantic, -c, -o, -E, -S**
 - `-std` - Установка стандарта
 - `-Wall` - Вывод сообщений о всех предупреждениях или ошибках, возникающих во время компиляции программы.
 - `-Werror` - Превращает все предупреждения в ошибки.
 - `-pedantic` - Выдаются все предупреждения, требуемые строгим ANSI стандартом С, отбрасываются все программы, которые используют запрещенные расширения
 - `-c` - скомпилировать
 - `-o "file"` - Поместить вывод в файл "file". Эта опция применяется вне зависимости от вида порождаемого файла, есть ли это выполнимый файл, объектный файл, ассемблерный файл или препроцессированный С код. Поскольку указывается только один выходной файл, нет смысла использовать '-o' при компиляции более чем одного входного файла, если вы не порождаете на выходе выполнимый файл. Если '-o' не указано, по умолчанию выполнимый файл помещается в 'a.out', объектный файл для 'исходный.суффикс' - в 'исходный.o', его ассемблерный код в 'исходный.s' и все препроцессированные С файлы - в стандартный вывод.
 - `-S` - Остановиться после собственно компиляции; не ассемблировать. Вывод производится в форме файла с ассемблерным кодом для каждого не ассемблерного входного файла. По умолчанию, имя файла с ассемблерным кодом делается из имени исходного файла заменой суффикса '.c', '.i', и.т.д. на '.s'. Входные файлы, которые не требуют компиляции игнорируются.
 - `-E` - Остановиться после стадии препроцессирования; не запускать собственно компилятор. Вывод делается в форме препроцессированного исходного кода, который посыпается на стандартный вывод. Входные файлы, которые не требуют препроцессирования игнорируются.

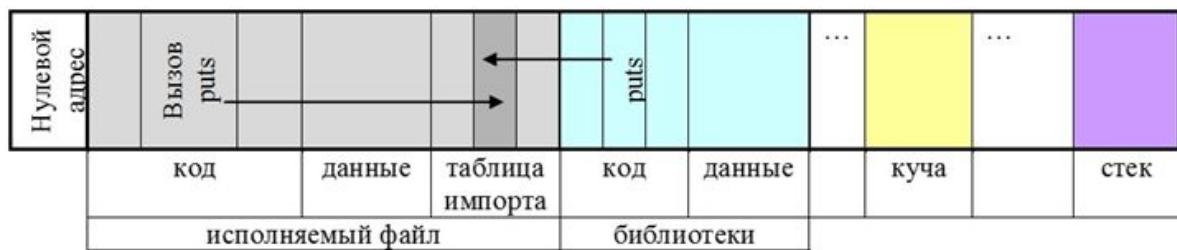
```
// 1. Препроцессор
gcc -E main.c > main.i
// 2. Трансляция на язык ассемблера
gcc -S main.i
// 3. Ассемблирование
gcc -c main.s
// 4. Компоновка
gcc -o main.exe main.o
```

```
// Вместо 1-3 можно написать  
gcc -c main.c
```

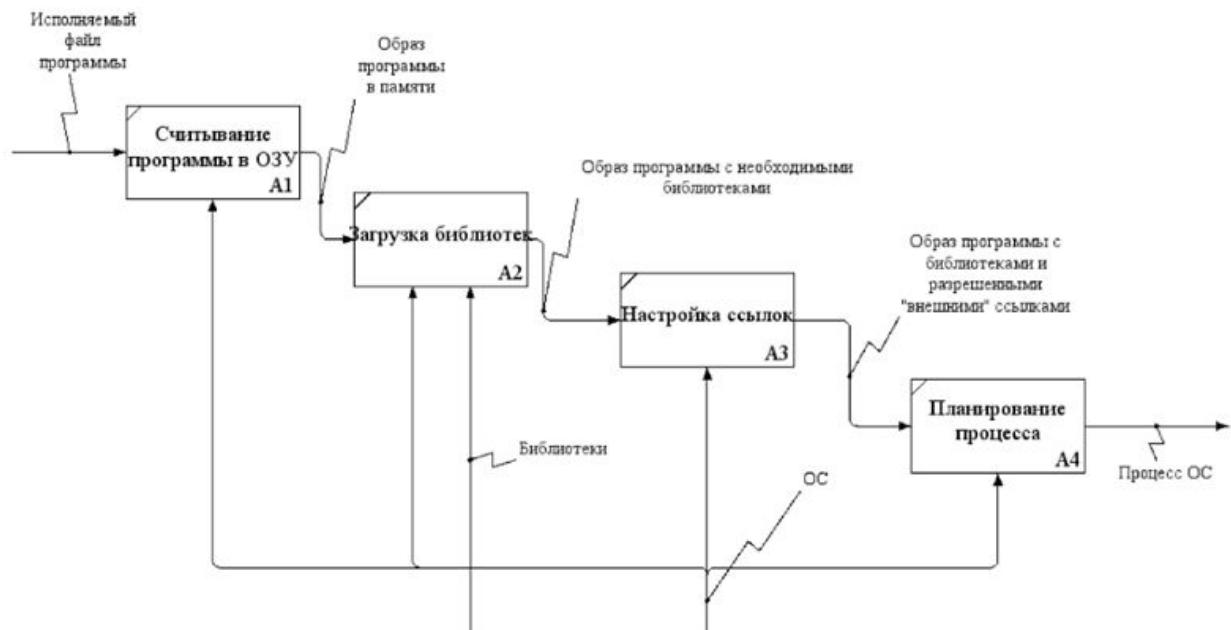
3. Исполняемый файл. Этапы запуска исполняемого файла. Функция main.

План ответа:

- **Формат исполняемого файла**
 - Отличная иллюстрация выше
- **Абстрактная память и процесс**



Запуск программы



- Заголовки функции **main** согласно стандарту C99
 - `int main(int argc, char *argv)`
 - `int main(void)`
 - `int main()`
- Возвращаемое значение функции **main**
(оператор: `return <значение>;`)

Это пришло из философии Юникса, где программа своим кодом возврата сообщает другим программам, успешно ли она отработала. 0 считается индикатором успешного выполнения, остальные значения -- нет.
- Аргумент функции **main**
`int main(int argc, char* argv[]) {...}`

Аргумент `argc` типа `integer` содержит в себе количество аргументов командной строки.

Аргумент `argv` типа `char` - указатель на массив строк. Каждый элемент массива указывает на аргументы командной строки. Один параметр отделяется от другого пробелами.

 - `argv[0]` - полное имя запущенной программы
 - `argv[1]` - первая строка записанная после имени программы
 - `argv[2]` - вторая строка записанная после имени программы
 - `argv[argc-1]` - последняя строка записанная после имени программы
 - `argv[argc]` - NULL

4. Переменные, операция присваивания, ввод/вывод значений переменных.

План ответа:

Понятие «переменная»

Переменная в программе представляет собой абстракцию ячейки памяти компьютера или совокупности таких ячеек.

Атрибуты переменной

Переменную можно охарактеризовать следующим набором атрибутов:

- Имя (идентификатор) – это строка символов, используемая для идентификации некоторой сущности в программе (переменной, функции и т. п.).
- Тип – определяет, как переменная хранится, какие значения может принимать и какие операции могут быть выполнены над переменной.
- Адрес – это ячейка памяти, с которой связана данная переменная. Адрес переменной иногда называют ее левым значением (l-value).
- Значение – это содержимое ячейки или ячеек памяти, связанных с данной переменной. Значение переменной иногда называется ее правым значением (r-value).
- Область видимости – это часть текста программы, в пределах которой переменная может быть использована.
- Время жизни – это интервал времени выполнения программы, в течение которого переменная существует (т.е. ей выделена память).

Описание переменных на языке Си

- Имя может содержать буквы, цифры и символ подчеркивания.
- Имя обязательно должно начинаться или с буквы, или с символа подчеркивания.
- Имя не должно совпадать с ключевыми словами языка.
- Имя чувствительно к регистру символов.
- Длина имени практически не ограничена.

Операция присваивания, особенности этой операции

Переменной может быть «назначено» значение с помощью присваивания.

```
width = 5;
length = 4;
speed = 25.34;      // speed = 25.34f;
```

Целым переменным обычно присваивают целые значения, вещественным – вещественные. Смешение типов возможно, но не всегда безопасно.

```
width = 17.5;
speed = 100;
```

После того как переменной присвоено значение, она может использоваться для вычисления значения другой переменной.

```
width = 5;  
length = 4;  
area = width * length;
```

Определение переменной можно совместить с присваиванием ей начального значения.

```
int width = 5;  
int length = 4;  
int area = width * length;
```

Функция printf

1. Синтаксис:

```
int printf(char* s, format, ...);
```

2. Аргументы:

format – указатель на строку с описанием формата.

3. Возвращаемое значение:

При успешном завершении вывода возвращается количество выведенных символов.

При ошибке возвращается отрицательное число.

4. Описание:

Функция printf выводит в стандартный поток вывода строку отформатированную в соответствии с правилами, указанными в строке, на которую указывает аргумент format.

Правила задаются набором трех типов директив:

- Обычные символы (кроме '%' и '\\'), которые выводятся без изменения
- Спецификаторы формата;
- Специальные символы.

Функция scanf

1. Синтаксис:

```
#include <stdio.h>
```

```
int scanf(const char *format, ...);
```

2. Аргументы:

format – указатель на строку с описанием формата.

3. Возвращаемое значение:

Возвращает количество успешно введенных данных

4. Описание:

Функция scanf() представляет собой процедуру ввода общего назначения, которая читает поток stdin и сохраняет информацию в переменных, перечисленных в списке аргументов. Она может читать все встроенные типы данных и автоматически преобразовывать их в соответствующий внутренний формат.

Управляющая строка, задаваемая параметром format, состоит из символов трех категорий:

- спецификаторов формата;
- пробельных символов;
- символов, отличных от пробельных.

Спецификации формата начинаются знаком % и сообщают функции scanf() тип данного, которое будет прочитано. Например, по спецификации %s будет прочитана строка, а по спецификации %d — целое значение. Стока форматирования читаются слева направо, и спецификации формата сопоставляются аргументам в порядке их перечисления в списке аргументов.

5. Операции и выражения.

План ответа:

Операция — конструкция в языках программирования, аналогичная по записи математическим операциям, то есть специальный способ записи некоторых действий.

Простейшим средством описания действий является выражение.

Выражение задает действия, которые вычисляют единственное значение. Состоит из констант, переменных, а также знаков операций и скобок.

Элементы данных, к которым применяется операция, называют операндами.

Каждая операция имеет операнды определенных типов и задает способ получения по значениям этих operandов нового значения определенного типа.

Некоторые операции имеют побочный эффект (side effect).

При выполнении этих операций, кроме основного эффекта - вычисления значения - происходят изменения объектов или файлов. Такова, например, операция постфиксного инкремента “*i++*”.

Каждая операция имеет определенный приоритет и ассоциативность. Это позволяет без лишних скобок однозначно понимать, к какой операции относится operand, стоящий между двумя соседними операциями.

Правила приоритетов операций при вычислении выражений определяют порядок, в котором выполняются операции, имеющие разные приоритеты.

$$a + b * c \Rightarrow a + (b * c)$$

На вопрос о том, какая из операций, имеющих разный приоритет, выполняется первой, отвечают правила ассоциативности.

$$a - b + c \Rightarrow ((a - b) + c) //$$

левоассоциативные $a = b = c \Rightarrow ((a = (b = c))) //$

право ассоциативные

Операция составного присваивания состоит из простой операции присваивания, скомбинированной с какой-либо другой бинарной операцией. При составном присваивании вначале выполняется действие, специфицированное бинарной операцией, а затем результат присваивается левому operandу. Выражение составного присваивания со сложением, например имеет вид: <выражение1> +=<выражение2>

Оно может быть записано и таким образом:

<выражение1> = <выражение1> + <выражение2>

Операции инкремента и декремента в C++

Операция	Обозначение	Пример	Краткое пояснение
преинкремент	<code>++</code>	<code>cout << ++value;</code>	Значение в переменной <code>value</code> увеличивается, после чего оператор <code>cout</code> печатает это значение
предекремент	<code>--</code>	<code>cout << --value;</code>	Значение в переменной <code>value</code> уменьшается, после чего оператор <code>cout</code> печатает это значение
постинкремент	<code>++</code>	<code>cout << value++;</code>	Оператор <code>cout</code> печатает значение переменной <code>value</code> , затем увеличивает это значение на 1
постдекремент	<code>--</code>	<code>cout << value--;</code>	Оператор <code>cout</code> печатает значение переменной <code>value</code> , затем уменьшает это значение на 1

6. Оператор-выражение, условный оператор и условная операция, составной оператор,

оператор `switch`. 

Выражения

формируют основные строительные блоки для операторов и определяют, каким образом программа управляет данными и изменяет их. Операторы определяют каким образом управление передается из одной части программы другой.

В языке Си любое выражение можно «превратить» в оператор, добавив к этому выражению точку с запятой:

`++i;`

В языке Си точка с запятой является элементом оператора и его завершающей частью, а не разделителем операторов.

Примеры:

`i = 1;` сохраняется в переменной `i`, затем значение операции (новое значение переменной `i`) вычисляется, но не используется.

`i—;` В качестве значения операции возвращается значение переменной `i`, оно не используется, но после этого значение переменной `i` уменьшается на 1.

`i * j - 1; // warning: statement with no effect` Поскольку переменные `i` и `j` не изменяются, этот оператор не имеет никакого эффекта и бесполезен.

Условный оператор

позволяет сделать выбор между двумя альтернативами, проверив значение

выражения. if (выражение) оператор_1 else оператор_2

Скобки вокруг выражения обязательны, они являются частью самого условного оператора. Часть else не является обязательной.

Не путайте операцию сравнения «==» и операцию присваивания «=» .

if ($i == 0$) НЕ эквивалентно if ($i = 0$) Чтобы проверить, что $i [0; n] \quad if (0 <= i \&\& i < n)$

...

Чтобы проверить противоположное условие $i [0; n] \quad if (i < 0 \parallel i >= n) ...$

Поскольку в выражении условного оператора анализируется числовое значение этого выражения, отдельные конструкции можно упростить. $if (\text{выражение} != 0)$

$if (\text{выражение})$

Составной оператор

В нашем «шаблоне» условного оператора указан только один оператор. Что делать, если нужно управлять несколькими операторами? Необходимо использовать составной оператор.

{

операторы

}

Заключая несколько операторов в фигурные скобки, мы заставляем компилятор интерпретировать их как один оператор.

if ($d > 0.0$)

{

$x_1 = (-b - \sqrt{d}) / (2.0 * a);$

$x_2 = (-b + \sqrt{d}) / (2.0 * a);$

}

Условная операция

состоит из двух символов «?» и «:», которые используются вместе следующим образом

```
expr_1 ? expr_2 : expr_3
```

Сначала вычисляется выражение expr_1. Если оно отлично от нуля, то вычисляется выражение expr_2, и его значение становится значением условной операции. Если значение выражение expr_1 равно нулю, то значением условной операции становится значение выражения expr_3.

```
#include <stdio.h> int  
main(void)  
{  
    int x = 5, y = 10, max = x > y ? x : y;  
  
    printf("Max of %d and %d is: %d\n", x, y, max); // Можно обойтись без  
    переменной max  
  
    printf("Max of %d and %d is: %d\n", x, y, x > y ? x : y);  
  
    return 0;  
}
```

Оператор switch

<pre>int mark = 4; if (mark == 5) printf("Excellent\n"); else if (mark == 4) printf("Good\n"); else if (mark == 3) printf("Averadge\n"); else if (mark == 2) printf ("Poor\n"); else printf("Illegal mark\n");</pre>	<pre>int mark = 4; switch (mark) { case 5: printf("Excellent\n"); break; case 4: printf("Good\n"); break; case 3: printf("Averadge\n"); break; case 2: printf("Poor\n"); break; default: printf("Illegal mark\n"); break; }</pre>
---	--

В общей форме оператор switch может быть записан следующим образом

```
switch (выражение)
```

```
{
```

```
    case константное_выражение : операторы,  
    ...
```

```
case константное_выражение : операторы  
default : операторы  
}
```

Управляющее выражение, которое располагается за ключевым словом switch, обязательно должно быть целочисленным (не вещественным, не строкой).

Константное выражение – это обычное выражение, но оно не может содержать переменных и вызовов функций.

5 константное выражение

5 + 10 константное выражение

n + 10 НЕ константное выражение

После каждого блока case может располагаться любое число операторов. Никакие скобки не требуются. Последним оператором в группе таких операторов обычно бывает оператор break.

Только одно константное выражение может располагаться в case-метке. Но несколько case-меток могут предшествовать одной и той же группе операторов.

Case-метки не могут быть одинаковыми. Порядок case-меток (даже метки default) не важен. Casemetka default не является обязательной.

Выполнение оператора break «внутри» оператора switch передает управление за оператор switch. Если бы оператор break отсутствовал, то стали бы выполняться операторы расположенные в следующих case-метках.

<pre>int mark = 4; switch (mark) { case 5: printf("Excellent\n"); case 4: printf("Good\n"); case 3: printf("Averadge\n"); case 2: printf("Poor\n"); default: printf("Illegal mark\n"); }</pre>	# На экране увидим Good Averadge Poor Illegal mark
---	--

7. Операторы цикла (while, do-while, for), операция запятая.

В языке Си цикл с предусловием реализуется с помощью оператора while. В общей форме этот оператор записывается следующим образом while (выражение) оператор

Выполнение оператора while начинается с вычисления значение выражения. Если оно отлично от нуля, выполняется тело цикла, после чего значение выражения вычисляется еще раз. Процесс продолжается в подобной манере до тех пор, пока значение выражения не станет равным 0.

```
#include <stdio.h>
int main(void)
{
    int sum, i, n = 5;
    // Сумма первых n натуральных чисел
    i = 1;
    sum = 0;
    while (i <= n)
    {
        sum += i;
        i++;
        // Можно обойтись одним оператором: sum += i++;
    }
    printf("Total of the first %d numbers is %d\n", n, sum);
    return 0;
}
```

В языке Си цикл с постусловием реализуется с помощью оператора do-while. В общей форме этот оператор записывается следующим образом do оператор while (выражение); Выполнение оператора do-while начинается с выполнения тела цикла. После чего вычисляется значение выражения. Если это значение отлично от нуля, тело цикла выполняется опять и снова вычисляется значение выражения. Выполнение оператора do-while заканчивается, когда значение этого выражения станет равным нулю.

```
#include <stdio.h>
int main(void)
{
    int digits = 0, n = 157;

    do
    {
        digits++;
        n /= 10;
    } while (n != 0);

    printf("The number has %d digit(s).\n", digits);

    return 0;
}
```

Оператор for обычно используют для реализации цикла со счетчиком. В общей форме этот оператор записывается следующим образом for (expr_1; expr_2; expr_3) оператор

Оператор цикла for может быть заменен (за исключением редких случаев) оператором while

<pre>expr_1; while (expr_2) { оператор expr_3; }</pre>	<p>expr_1 – шаг инициализации, который выполняется только один раз.</p> <p>expr_2 – выражение отношения или логическое выражение. Управляет завершением цикла.</p> <p>expr_3 – выражение, которое выполняется в конце каждой итерации цикла.</p>
--	--

Пример:

<pre>#include <stdio.h> int main(void) { int sum, i, n = 5; i = 1; sum = 0; while (i <= n) { sum += i; i++; } }</pre>	<pre>#include <stdio.h> int main(void) { int sum = 0, i, n = 5; for (i = 1; i <= n; i++) sum += i; ... }</pre>
--	--

Любое из трех выражений expr_1, expr_2, expr_3 можно опустить, но точки с запятой должны остаться на своих местах. Если опустить expr_1 или expr_3, то соответствующие действия выполнятся не будут.

```
i = 1;  
for ( ; i <= n; )  
    sum += i++;
```

Если же опустить проверку условия expr_2, то по умолчанию считается, что условие продолжения цикла всегда истинно.

```
for ( ; ; )  
    printf("Infinity loop\n");
```

Идиомы:

```
// Считать в прямом направлении от 0 до n-1  
    for (i = 0; i < n; i++) ...  
  
// Считать в прямом направлении от 1 до n  
    for (i = 1; i <= n; i++) ...  
  
// Считать в обратном направлении от n-1 до 0  
    for (i = n-1; i >= 0; i--) ...  
  
// Считать в обратном направлении от n до 1  
    for (i = n; i > 0; i--) ...
```

В C99 первое выражение expr_1 в цикле for может быть заменено определением. Эта особенность

позволяет определять переменные для использования в цикле

```
for (int i = 0; i < n; i++)
```

Переменную *i* не нужно объявлять до оператора *for*.

```
for (int i = 0; i < n; i++)
```

```
{
```

```
...
```

```
printf("%d", i); // OK
```

```
...
```

```
}
```

```
printf("%d", i); // ОШИБКА
```

Иногда бывает необходимо написать оператор *for* с двумя или более выражениями инициализации или изменить несколько переменных в конце цикла. Это можно сделать с помощью операции запятая. выражение_1, выражение_2

Эта операция выполняется следующим образом: сначала вычисляется выражение_1 и его значение отбрасывается, затем вычисляется выражение_2. Значение этого выражения является результатом операции всей операции.

выражение_1 всегда должно содержать побочный эффект. В противном случае от этого выражения не будет никакого толка.

```
for (sum = 0, i = 1, n = 5; i <= n; i++, sum += i)
```

```
; // пустой оператор
```

8. Операторы *break*, *continue*, *goto*. Пустой оператор.

Оператор **break** может использоваться для принудительного выхода из циклов *while*, *do-while* и *for*. Выход выполняется из ближайшего цикла или оператора *switch*.

```
for (d = 2; d < n; d++)
```

```
    if (n % d == 0)
```

```
        break;
```

```
    if (d < n)
```

```
        printf("%d is divisible by %d\n", n, d);
```

```
    else
```

```
        printf("%d is prime\n", n);
```

Оператор **continue** передает управление в конец цикла.

В циклах while и do-while это означает переход к проверке управляющего выражения, а в цикле for – выполнение expr_3 и последующую проверку expr_2.

Оператор continue может использоваться только внутри циклов.

```
sum = 0;
i = 0;
while (i < 10)
{
    scanf("%d", &num);
    if (num < 0)
        continue;
    sum += num;
    i++;
}
```

Оператор **goto** способен передать управление на любой оператор (в отличие от операторов break и continue) функции, помеченный меткой.

Метка – это идентификатор, расположенный вначале оператора:

идентификатор : оператор

Оператор может иметь более одной метки. Сам оператор goto записывается в форме
goto идентификатор;

```
#include <stdio.h>

// Определение "простоты" числа
int main(void)
{
    int d, n = 17;

    for (d = 2; d < n; d++)
        if (n % d == 0)
            goto done;

done:
    if (d < n)
        printf("%d is divisible by %d\n", n, d);
    else
        printf("%d is prime\n", n);

    return 0;
}
```

Считается, что оператор goto источник потенциальных неприятностей. Этот оператор на практике практически никогда не бывает необходим и почти всегда легче обходится без него.

Есть несколько ситуаций, в которых без goto удобно использовать. Например, когда необходимо сразу выйти из двух и более вложенных циклов.

Пустой оператор состоит только из символа «;». Основная «специализация» пустого оператора – реализация циклов с пустым телом:

```
for (d = 2; d < n; d++)      for (d = 2; n % d != 0 && d < n; d++)
    if (n % d == 0)           ;
        break;
```

Пустой оператор легко может стать источником ошибки:

```
if(d == 0);
printf("ERROR: division by zero!\n");
```

9. Функции.

План ответа:

Подпрограмма - именованная часть программы, содержащая описание определённого набора действий. Подпрограмма может быть многократно вызвана из разных частей программы.

Функция - это подпрограмма специального вида, которая всегда должна возвращать результат. Вызов функции является, с точки зрения языка программирования, выражением, он может использоваться в других выражениях или в качестве правой части присваивания.

Процедура - это независимая именованная часть программы, которую после однократного описания можно многократно вызвать по имени из других частей программы для выполнения определенных действий.

Преимущества подпрограмм

- Модульность (использование подпрограмм позволяет разделить большую программу на части меньшего размера, которые легче понимать и изменять).
- Закрытость реализации (изменение реализации подпрограммы не влияет на остальную часть программы, если интерфейс подпрограммы не изменился).
- Расширение возможностей языков программирования (создание библиотек).

В языке Си подпрограммы представлены только функциями.

```
// заголовок функции
тип-результата имя-функции (список формальных параметров
                                с их типами)

// тело функции
{
    определения
    операторы
}

float avg(float a, float b)
{
    float c;
    c = a + b;
    return c / 2.0;
}
```

- **оператор return;**

- Завершает выполнение функции и возвращает управление вызывающей стороне.
- Используется для возврата значения (если функция возвращает результат).
- Функция может содержать произвольное число операторов return.
- Оператор return может использоваться в функциях типа void. При этом никакое выражение не указывается.

- **операция вызова функции;**

Для вызова функции необходимо указать ее имя, за которым в круглых скобках через запятую перечислить аргументы.

```
float a = avg(2.0, 5.0);
```

Если функция возвращает значение, ее можно использовать в выражениях.

```
float a, b;

printf("Enter a and b:");
scanf("%f%f", &a, &b);

if (avg(a, b) < 0.0)
    printf("Averadge is negative!\n");
```

- **объявление и определение функции;**

```
#include <stdio.h>

float avg(float a, float b); // float avg(float, float);

int main(void)
{
    float a = avg(2.0, 3.0);

    printf("%f\n", a);

    return 0;
}

float avg(float a, float b)
{
    return (a + b) / 2.0;
}
```

- передача аргументов в функцию;

В Си все аргументы функции передаются «по значению».

Авторы языка: «Благодаря этому свойству обычно удастся написать более компактную программу, содержащую меньшее число посторонних переменных, поскольку параметры можно рассматривать как должным образом инициализированные локальные переменные.»

- рекурсия.

Функция называется *рекурсивной*, если она вызывает саму себя.

Например, следующая функция рекурсивно вычисляет факториал, используя формулу $n! = n * (n - 1)!$

```
int fact(int n)
{
    if (n == 0)
        return 1;

    return n * fact(n - 1);
}
```

10. Автоматизация сборки проекта, утилита make. Сценарий сборки проекта. Простой сценарий сборки. Использование переменных и комментариев.

Сборка программы с разными параметрами компиляции. 

План ответа:

- **автоматизация сборки проекта: основные задачи, «исходные» данные;**
 1. Задача автоматизации сборки проекта – избавить программиста от необходимости каждый раз печатать объёмные вызовы компилятору и компоновщику в весьма больших проектах.
 2. Данными для автоматической сборки являются файлы заголовков, реализации и библиотеки (вход), исполняемые файлы и библиотеки (выход)
- **разновидности утилиты make;**

make — утилита, автоматизирующая процесс преобразования файлов из одной формы в другую.

- GNU Make (рассматривается далее)
- BSD Make
- Microsoft Make (nmake)

- **сценарий сборки проекта: название файла, его структура;**

```
цель: зависимость_1 ... зависимость_n
[tab]команда_1
[tab]команда_2
...
[tab]команда_m
```

- **простой сценарий сборки;**

Простой сценарий сборки

```
greeting.exe : hello.o buy.o main.o
    gcc -o greeting.exe hello.o buy.o main.o

test_greeting.exe : hello.o buy.o test.o
    gcc -o test_greeting.exe hello.o buy.o test.o

hello.o : hello.c hello.h
    gcc -std=c99 -Wall -Werror -pedantic -c hello.c

buy.o : buy.c buy.h
    gcc -std=c99 -Wall -Werror -pedantic -c buy.c

main.o : main.c hello.h buy.h
    gcc -std=c99 -Wall -Werror -pedantic -c main.c

test.o : test.c hello.h buy.h
    gcc -std=c99 -Wall -Werror -pedantic -c test.c

clean :
    rm *.o *.exe
```

- **использование переменных;**

Определить переменную в make-файле можно следующим образом:

```
VAR_NAME := value
```

Чтобы получить значение переменной, необходимо ее имя заключить в круглые скобки и перед ними поставить символ '\$'.

```
$ (VAR_NAME)
```

- **условные конструкции в сценарии сборки.**

- ifeq(op1, op2) oper1 [else oper2] endif, ifneq(op1, op2) oper1 [else oper2] endif

11. Автоматизация сборки проекта, утилита make. Сценарий сборки проекта.

Автоматические переменные. Шаблонные правила. 

План ответа:

- **автоматизация сборки проекта: основные задачи, «исходные» данные**

- **основные задачи:**

1. Задача автоматизации сборки проекта – избавить программиста от необходимости каждый раз печатать объёмные вызовы компилятору и компоновщику в весьма больших проектах.

2. Данными для автоматической сборки являются файлы заголовков, реализаций и библиотеки (вход), исполняемые файлы и библиотеки (выход).

- **исходные данные:**

- make file

- исходники проекта
- **разновидности утилиты make**
 - GNU Make (рассматривается далее)
 - BSD Make
 - Microsoft Make (nmake)

- сценарий сборки проекта: название файла, его структура

```
цель: зависимость_1 ... зависимость_n
[tab]команда_1
[tab]команда_2
...
[tab]команда_m
```

- автоматические переменные

Автоматические переменные - это переменные со специальными именами, которые «автоматически» принимают определенные значения перед выполнением описанных в правиле команд.

- Переменная "\$^" означает "список зависимостей".
- Переменная "\$@" означает "имя цели".
- Переменная "\$<" является просто первой зависимостью.

Было

```
greeting.exe : $(OBJS) main.o
gcc -o greeting.exe $(OBJS) main.o
```

Стало

```
greeting.exe : $(OBJS) main.o
gcc $^ -o $@
```

- шаблонные правила

Шаблонные правила

```
%_.расш_файлов_целей : %.расш_файлов_зав
[tab]команда_1
[tab]команда_2
...
[tab]команда_m

# Компилятор
CC := gcc

# Опции компиляции
CFLAGS := -std=c99 -Wall -Werror -pedantic

# Общие объектные файлы
OBJS := hello.o buy.o

greeting.exe : $(OBJS) main.o
$(CC) $^ -o $@

test_greeting.exe : $(OBJS) test.o
$(CC) $^ -o $@

%.o : %.c *.h
$(CC) $(CFLAGS) -c $<

clean :
$(RM) *.* *.exe
```

12. Типы языка Си. Преобразование типов.

План ответа:

Тип данных - множество значений и операций на этих значениях

классификация типов языка Си:

Простые (скалярные) типы

целый (int);

вещественный (float и др.);

символьный (char);

перечисляемый тип;

логический тип (c99);

void;

указатели.

Составные (строктурированные) типы

массивы;

структуры;

объединения.

Целый тип:

В языке Си существует несколько типов целых чисел. Они различаются

- объемом памяти, отводимым под переменную (диапазоном);
- возможностью присваивания положительных и отрицательных чисел.

Для создания нужного целого типа используется ключевое слово int и модификаторы типа:

- signed
- unsigned
- short
- long

Тип	Наименьшее значение	Наибольшее значение	Размер в байтах
short int	-32786	32767	2
unsigned short int	0	65535	2
int	-2147483648	2147483647	4
unsigned int	0	4294967295	4
long int	-2147483648	2147483647	4
unsigned long int	0	4294967295	4
long long int	-9223372036854775808	9223372036854775807	8
unsigned long long int	0	18446744073709551617	8

Вещественный тип:

- float
- double
- long double

Символьный тип:

Для работы с символами предназначен тип `char`. (Под набором символов, как правило, понимается набор символов ASCII.)

Переменной типа `char` может быть присвоено значение любого ASCII символа.

```
char ch;
ch = 'a';
ch = 'A';
ch = '0';
ch = ' ';
```

Под значение типа `char` отводится один байт.

Стандарт не определяет «знаковость» этого типа. Если нужно подчеркнуть «знаковость», можно использовать модификаторы `signed` и `unsigned` с типом `char`.

```
signed char sch;
unsigned char uch;
```

Для ввода и вывода значений символьного типа используется спецификатор `%c`.

```
char ch;

scanf("%c", &ch);
printf("%c %d", ch, ch);
```

Перечисляемый тип:

- Тип разработан для переменных, которые принимают небольшое количество значений.
- Значения этого типа перечисляются программистом.

```
// Отдельное описание типа
enum SEASONS {WINTER, SPRING, SUMMER, AUTUMN};

...
{

SEASONS s;
```

```
// Описание типа совмещено с определением переменной
enum {WINTER, SPRING, SUMMER, AUTUMN} s;
```

Си трактует значения перечислимого типа как целые числа.

```
enum SEASONS {WINTER = 1, SPRING = 2, SUMMER = 3, AUTUMN = 4};

int i;
enum {WINTER, SPRING, SUMMER, AUTUMN} season;

i = WINTER;           // 0
season = 0;           // 0 (WINTER)
season++;             // 1 (SPRING)
i = season + 2;      // 3

season = 4;           // 4 (?)
```

Логический тип (C99)

Стандарт C99 добавил логический тип _Bool.

```
bool flag;
```

Переменные типа _Bool могут принимать только значения 0 и 1.

```
flag = 5;
printf("%d", flag); // 1
```

Стандарт c99 предоставляет заголовочный файл stdbool.h, который облегчает использование «нового» логического типа.

```
#include <stdbool.h>
...
bool flag;
...
flag = true;
```

Void

В заголовке функции void может указывать на то, что функция не возвращает значение;
функция не имеет параметров.

void* - указатель, способный представлять адреса любых объектов.

оператор typedef;

Позволяет определять имена новых типов.

typedef тип имя;

“+” улучшает читаемость.

“+” облегчает внесение изменений.

```
unsigned int n_pens, n_copybooks;

typedef unsigned int quantity_t;

quantity_t n_pens, n_copybooks;
```

Операция sizeof

возвращает размер переменной или типа в байтах .
sizeof(выражение);

```

char ch;
int i;

printf("%d\n", sizeof(ch));      // 1
printf("%d\n", sizeof i);       // 4
printf("%d\n", sizeof(double)); // 8

```

Операция	Название	Нотация	Класс	Приоритет	Ассоциат.
sizeof	Размер	sizeof X	Префиксная	15	Справа налево

неявное и явное преобразование типов.

13. Статические одномерные массивы.



- понятие «массив»

- последовательность элементов одного и того же типа, расположенных в памяти друг за другом.

- определение переменной-массива, способы инициализации переменной-массива

- тип элемента может быть любым
- количество элементов указывается целочисленным константным выражением
- количество элементов не может быть изменено в ходе выполнения программы

```
#define N 10
```

```
...
```

```
int a[N];
```

- операция индексации

- для доступа к элементу массива используется индекс.
- индексация выполняется с нуля.
- в качестве индекса может выступать целочисленное выражение
- Си не предусматривает никаких проверок на выход за пределы массива.

- особенности использования массивов в языке Си

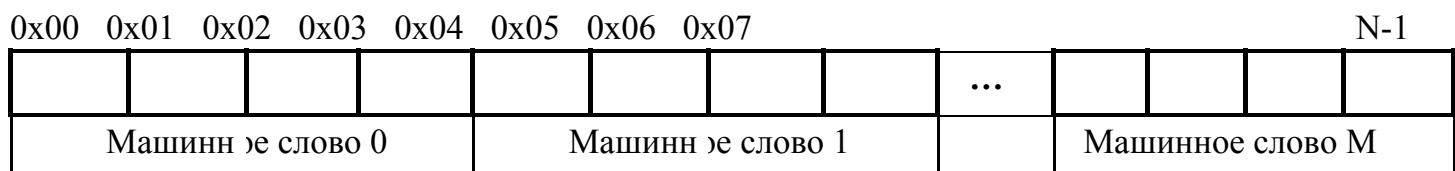
- компилятор может самостоятельно определить количество элементов в массиве и выделить для них память
- массив
- есть указатель на первый элемент
- адресная арифметика - не предусмотрено проверок на выход за пределы массива

- индексом может служить целочисленное выражение
 - количество элементов статического массива не может быть изменено в ходе выполнения программы
- массивы как параметры функции**
- передается массив в функцию по указателю на начало и конец, указателю на начало и количеству элементов

14. Указатели, базовые операции.

План ответа:

Организация памяти с точки зрения программиста.



Понятие «указатель».

- Указатель – это объект, содержащий адрес объекта или функции, либо выражение, обозначающее адрес объекта или функции.

Разновидности указателей в языке Си.

- Типизированный указатель на данные (тип* имя)
- Бестиповой указатель (void* имя)
- Указатель на функцию

Использование указателей.

- Передача параметров в функцию: изменяемые параметры, «объемные» параметры
- Обработка областей памяти: динамическое выделение памяти, ссылочные структуры данных

Определение переменной-указателя.

- При определении переменной-указателя перед ее именем должен размещаться символ '*'.

Базовые операции над указателями (“&” и “*”).

Для определения адреса данных используется операция ‘&’ (операция взятия адреса).

Для доступа к данным, на которые указывает указатель, используется операция '*' (операция разыменования).

Пример:

```
int a = 1;  
int *p = &a; // p теперь указывает на a
```

Операции '&' и '*' взаимно обратные.

```
int d = 5, *p = &d;  
*&d === d  
&*p === p
```

Модификатор `const` и указатель.

```
Указатель на константу int a  
= 5; int b = 7;  
const int *p = &a; // указатель на константу int * const p  
= &a; // константный указатель  
const int *const p = &a; // константный указатель на константу
```

15. Указатели, массивы, адресная арифметика.

План ответа:

Понятие «указатель».

Результат выражения, состоящего из имени массива, представляет собой адрес области памяти, выделенной под этот массив.

Связь между указателями и статическими массивами.

- Операция `sizeof` для массива

```
int a[10];  
int *pa = a; // sizeof(a) = 40, sizeof(pa) = 4
```

- Массив – operand операции получения адреса: `&a` возвращает указатель, НО тип этого указателя - указатель на массив (не указатель на целое)

- Строковый литерал-инициализатор массива

```
char[] char a[] = "abcdef"; // sizeof(a) = 7
```

- Отличие массивов и указателей:

- При определении массива и указателя под соответствующие переменные выделяется разное количество памяти
- Выражению из имени массива нельзя присвоить другое значение

Адресная арифметика (сложение указателя с числом, сравнение указателей, вычитание указателей).

Сложение указателя с числом:

тип *p = ...;

p += n;

Новый адрес в p = старый адрес из p + n * sizeof(тип)

p -= m;

Новый адрес в p = старый адрес из p - m * sizeof(тип)

int a[10];

&a[i] == a + i

Выражение a[i] компилятор заменяет на *(a + i)

Сравнение указателей: при сравнении указателей сравниваются адреса. При этом можно сравнивать указатель с NULL, сравнивать два однотипных указателя.

Вычитание указателей: тип a[10];

тип *pa = a, *pb = &a[2];

pb - pa = (адрес из pb - адрес из pa) / sizeof(тип)

16. Указатели, void*, указатели на функции (на примере функции qsort).

План ответа:

void*, особенности операций с ним;

- Тип указателя void используется, если тип объекта неизвестен.
- позволяет передавать в функцию указатель на объект любого типа;

- полезен для ссылки на произвольный участок памяти, независимо от размещенных там объектов.
- В языке С допускается присваивание указателя типа `void*` указателю любого другого типа (и наоборот) без явного преобразования типа указателя.

```
double d = 5.0;
double *pd = &d;
void *pv = pd;
pd = pv;
```

- Указателя типа `void*` нельзя разыменовывать.
- К указателям типа `void*` не применима адресная арифметика.

приведение указателей разных типов к `void*` и обратно;

определение указателя на функцию;

- `double trapezium(double a, double b, int n,`
`double (*func)(double));`

присваивание значения указателю на функцию;

- `result = integrate(0, 3.14, 25, sin);`

вызов функции по указателю;

- `y = (*func)(x); // y = func(x);`

использование указателей на функции.

- `void qsort(void *base, size_t nmemb, size_t size,`
`int (*compar)(const void*, const void*));`
- Пусть необходимо упорядочить массив целых чисел по возрастанию.
`int compare_int(const void* p, const void* q)`
`{`
`const int *a = p;`
`const int *b = q;`
`return *a - *b; // return *(int*)p - *(int*)q;`
`}`
`...`

```
int a[10];
...
qsort(a, sizeof(a) / sizeof(a[0]), sizeof(a[0]),
       compare_int)
```

17. Динамические одномерные массивы.

План ответа:

- **Функции для выделения и освобождения памяти (malloc, calloc, realloc, free). Порядок работы и особенности использования этих функций**
 - Память, которую использует программа делится на три вида:
 - Статическая память (static memory)
 - хранит глобальные переменные и константы
 - размер определяется при компиляции
 - Стек (stack)
 - хранит локальные переменные, аргументы функций и промежуточные значения вычислений
 - размер определяется при запуске программы (обычно выделяется 4 Мб)
 - Куча (heap)
 - динамически распределяемая память
 - ОС выделяет память по частям (по мере необходимости)
 - Прототипы функций:
 - `void *malloc(size_t size);`
 - `void free(void *ptr);`
 - `void *calloc(size_t nmemb, size_t size);`
 - `void *realloc(void *ptr, size_t size);`
- **Типичные ошибки при работе с динамической памятью (утечка памяти, «дикий» указатель, двойное освобождение).**
 - Двойное освобождение
 - `free(p);`
 - `free(p); // Segmentation fault`
 - Дикий указатель - указатель на объект, которого не существует, или который был потерян в процессе выполнения программы.
 - Утечка памяти - процесс в программе, когда память занята большим количеством вызовов функций calloc, malloc или realloc, но не освобождена функцией free.

18. Указатели и многомерные статические массивы.



План ответа:

- **концепция многомерного массива как «массива массивов»**
 - Многомерный массив является по своей структуре массивом, состоящим из элементов, каждый из которых является также массивом. В памяти элементы многомерного массива располагаются последовательно друг за другом.
 - $a_{00} \ a_{01} \ \dots \ a_{0N}$
...
 $a_{M0} \ a_{M1} \ \dots \ a_{MN}$
- **определение многомерных массивов**

```
#define M 2
#define N 3

int main(void)
{
    int a[M][N];
    int a[2][3]; // Создание целочисленного двумерного массива, в котором 2 строки и 3 столбца
    float a[2][3];
    char a[2][3];
    int c[2][2] = {[0][0] = 1, [1][1] = 1}; // C.99

    return 0;
}
```

- **инициализация многомерных массивов**

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

int d[][2] = { {1, 2} };

int e[][] =
{
    {1, 2},
    {4, 5}
```

```
};

// error: array type has incomplete element type
int b[3][3] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
// warning: missing braces around initializer [-Wmissing-braces]
// c99
int c[2][2] = {[0][0] = 1, [1][1] = 1};
```

- **«слои», составляющие многомерные массивы**
- **обработка многомерных массивов с помощью указателей**

Обработка строки матрицы (обнуление i-ой строки)

```
// указатель на начало i-ой строки
int *p = &a[i][0];
&a[i][0] => &(*(a[i] + 0)) => &(*a[i]) => a[i]
```

Т.е. выражение $a[i]$ – это адрес начала i-ой строки.

```
// обнуление i-ой строки
for (p = a[i]; p < a[i] + M; p++)
    *p = 0;
```

$a[i]$ можно передать любой функции, обрабатывающей одномерный массив:

```
zero(a[i], M);
```

Обработка столбца матрицы (обнуление j-го столбца)

```
// указатель на строку (строка – это массив из M элементов)
int (*q)[M];
```

Скобки важны из-за приоритета операций! Без скобок получится массив из M указателей.

Выражение $q++$ смещает указатель на следующую строку

Выражение $(*q)[j]$ возвращает значение в j-ом столбце строки, на которую указывает q.

```
for (q = a; q < a + N; q++)
    (*q)[j] = 0;
```

- **передача многомерных массивов в функцию**

Пусть определена матрица

```
int a[N][M];
```

Для ее обработки могут быть использованы функции со следующими прототипами:

```
void f(int a[N][M], int n, int m);
void f(int a[][M], int n, int m);
void f(int (*a)[M], int n, int m);
```

Неверные прототипы:

```
void f(int a[][], int n, int m);
// не указана вторая размерность => компилятор не сможет
// выполнить обращение по индексу
void f(int *a[M], int n, int m);
void f(int **a, int n, int m);
// неверный тип – массив указателей
```

- **const и многомерные массивы.**

```
void print(const int arr[][][M], int n, int m)
{
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < m; j++)
            printf("%d ", arr[i][j]);
        printf("\n");
    }
    printf("\n");
}

...
int a[N][M] = {{1, 2, 3}, {4, 5, 6}};
// expected 'const int (*)[5]' but argument is of type 'int(*)[5]'
print(a, 2, 3);
```

Формальное объяснение

Согласно C99 6.7.3 #8 и 6.3.2.3.2 выражение $T(*p)[N]$ не преобразуется неявно в $T \text{ const } (*p)[N]$.

•

Способы борьбы

– не использовать `const`;

– использовать явное преобразование типа

```
print((const int (*)[M]) a, 2, 3);
```

Почему такое неявное преобразование запретили

```
const char c = 'x';           /* 1 */
```

```
char *p1; /* 2 */
// warning: initialization from incompatible pointer type
const char **p2 = &p1; /* 3 */
*p2 = &c; /* 4 */
*p1 = 'X'; /* 5 */
```

3: В p2 поместили адрес p1.

4: С помощью p2 изменили p1. p1 теперь указывает на с.

5: С помощью p1 изменили константу.

19. Массивы переменной длины (c99), их преимущества и недостатки, особенности использования.

Стандарт C99 позволяет в объявлении размера массива использовать любые выражения, в том числе такие, значение которых становится известным только во время выполнения. Объявленный таким образом массив называется *массивом переменной длины*. Приведем пример массива переменной длины:

```
void f(int dim)
{
    char str[dim]; /* символьный массив переменной длины */
    /* ... */
}
```

Здесь размер массива str определяется значением переменной dim, которая передается в функцию f() как параметр. Таким образом, при каждом вызове f() создается массив str разной длины.

Преимущества и недостатки:

- переменную длину могут иметь только локальные массивы

20. Динамические многомерные массивы.

План ответа:

- Способы выделения памяти для динамических матриц: идеи, реализации, анализ преимуществ и недостатков.

- Матрица как одномерный массив

```
double *data;  
int n = 3, m = 2;  
data = malloc(n * m * sizeof(double));  
if (data)  
{  
    for (int i = 0; i < n; i++)  
        for (int j = 0; j < m; j++)  
            // Обращение к элементу i, j  
            data[i*m+j] = 0.0;  
    free(data);  
}
```

Преимущества:

1. Простота выделения и освобождения памяти.
2. Возможность использовать как одномерный массив.

Недостатки:

1. Средство проверки работы с памятью (СПРП), например Doctor Memory, не может отследить выход за пределы строки.
2. Нужно писать $i * m + j$, где m – число столбцов

- Матрица как массив указателей

```
void free_matrix_rows(double **data, int n);  
double** allocate_matrix_rows(int n, int m)  
{  
    double **data = calloc(n, sizeof(double*));  
    if (!data)  
        return NULL;  
    for (int i = 0; i < n; i++)  
    {  
        data[i] = malloc(m * sizeof(double));  
        if (!data[i])  
        {  
            free_matrix_rows(data, n);  
            return NULL;  
        }  
    }  
}
```

```
    return data;  
}
```

Преимущества:

1. Возможность обмена строки через обмен указателей.
2. СПРП может отследить выход за пределы строки.

Недостатки:

1. Сложность выделения и освобождения памяти.
2. Память под матрицу "не лежит" одним куском.

- **Объединение подходов**

```
double** allocate_matrix_solid(int n, int m)  
{  
    double **data = malloc(n * sizeof(double*) +  
                           n * m * sizeof(double));  
    if (!data)  
        return NULL;  
  
    for (int i = 0; i < n; i++) {  
        data[i] = (double*)((char*) data +  
                           n * sizeof(double*) + i * m * sizeof(double));  
    }  
    return data;  
}
```

Преимущества:

1. Простота выделения и освобождения памяти.
2. Возможность использовать как одномерный массив.
3. Перестановка строк через обмен указателей.

Недостатки:

1. Сложность начальной инициализации.
2. СПРП не может отследить выход за пределы строки.

21. Строки.

План ответа:

- **понятия «строка» и «строковый литерал»**

- Стока – это последовательность символов, заканчивающаяся и включающая первый нулевой символ (англ., null character ‘\0’ (символ с кодом 0)).
- Строковый литерал – последовательность символов, заключенных в двойные кавычки.

```
char str[] = "String for test";
printf("Max is %d\n", max);
```

- **определение переменной-строки, инициализация строк;**

Определение переменной-строки, которая может содержать до 80 символов обычно выглядит следующим образом:

```
#define STR_LEN 80
...
char str[STR_LEN+1]; // !
```

Поскольку строка – массив символов, для доступа к элементу строки может использоваться операция индексации:

```
int count_spaces(const char *s)
{
    int count = 0;
    for (int i = 0; s[i] != '\0'; i++)
        if (s[i] == ' ')
            count++;
    return count;
}
```

- **ввод/вывод строк (scanf, gets, fgets, printf, puts);**

Функции scanf и gets небезопасны и недостаточно гибки. Программисты часто реализуют свою собственную функцию для ввода строки, в основе которой лежит посимвольное чтение вводимой строки с помощью функции getchar.

```
char *fgets(char *s, int size, FILE *stream);
```

Прекращает ввод когда (любое из)

- прочитан символ ‘\n’;
- достигнут конец файл;
- прочитано size-1 символов.

Введенная строка всегда заканчивается нулем.

- ```
fgets(str, sizeof(str), stdin);
```
- **функции стандартной библиотеки для работы со строками (strcpy, strlen, strcmp и др.);**  
// Они и так понятны из имени и можно догадаться без проблем какие имеют прототипы
- 

## 22. Область видимости, время жизни и связывание.

План ответа:

- **понятия «область видимости», «время жизни» и «связывание» в языке Си**
  - область видимости (scope) имени – это часть текста программы, в пределах которой имя может быть использовано
    - блок
    - файл
    - функция
    - прототип функции
  - блок
    - переменная, определенная внутри блока, имеет область видимости в пределах блока.
    - формальные параметры функции имеют в качестве области видимости блок, составляющий тело функции.
  - файл
    - область видимости в пределах файла имеют имена, описанные за пределами какой бы то ни было функции.
    - переменная с областью видимости в пределах файла видна на протяжении от точки ее описания и до конца файла, содержащего это определение. - имя функции всегда имеет файловую область видимости.
  - функция
    - метки - это единственные идентификаторы, область действия которых
    - функция.
    - метки видны из любого места функции, в которой они описаны. - в пределах функции имена меток должны быть уникальными.
  - прототип функции

- область видимости в пределах прототипа функции применяется к именам переменных, которые используются в прототипах функций.
- область видимости в пределах прототипа функции простирается от точки, в которой объявлена переменная, до конца объявления прототипа
- время жизни (storage duration) – это интервал времени выполнения программы, в течение которого «программный объект» существует.
  - глобальное (по стандарту - статическое (англ. static))
  - локальное (по стандарту - автоматическое (англ. automatic))
  - динамическое (по стандарту - выделенное (англ. allocated))
- связывание (linkage) определяет область программы (функция, файл, вся программа целиком), в которой «программный объект» может быть доступен другим функциям программы
  - внешнее (external)
  - внутреннее (internal)
  - никакое (none)

#### **- правила перекрытия областей видимости**

- переменные, определенные внутри некоторого блока, будут доступны из всех блоков, вложенных в данный
- возможно определить в одном из вложенных блоков переменную с именем, совпадающим с именем одной из "внешних" переменных

#### **- размещение «объектов» в памяти в зависимости от времени жизни**

- переменные
  - глобальные - статические
  - локальные - автоматические
  - динамические - выделенные
- класс памяти auto
  - применим к переменным, определенным в блоке
  - локальное время жизни, видимость в пределах блока и не имеет связывания
  - любая переменная, объявлена в блоке или заголовке функции
- класс памяти static

- может использоваться с любыми переменными независимо от места их расположения
  - для переменной все какого либо блока static изменяет связывание этой переменной на внутреннее
    - для переменной в блоке изменяет время жизни с автоматического на глобальное - статическая переменная, определенная вне какого
    - либо блока имеет глобальное время жизни, область видимости в пределах файла и внутреннее связывание
      - скрывает переменную в файле, в котором она определена
      - статическая переменная, определенная в блоке имеет глобальное время жизни, область видимости в пределах блока и отсутствие связывания
        - такая переменная сохраняет свое значение после выхода из блока.
        - инициализируется только один раз.
        - если функция вызывается рекурсивно, это порождает новый набор локальных переменных, в то время как статическая переменная разделяется между всеми вызовами.
- класс памяти extern
  - помогает разделить переменную между несколькими файлами
  - используется для переменных, определенных как в блоке, так и вне блока
    - объявлений (extern int number;) может быть сколько угодно.
    - определение (int number;) должно быть только одно.
    - объявления и определение должны быть одинакового типа.
  - глобальное время жизни, файловая область видимости
- класс памяти register
  - использование класса памяти register – просьба (!) к компилятору разместить переменную не в памяти, а в регистре процессора
    - используется только для переменных, определенных в блоке.
    - задает локальное время жизни, видимость в блоке и отсутствие связывания.
    - обычно не используется.
  - к переменным с классом памяти register нельзя применять операцию получения адреса &
- **влияние связывания на объектный и/или исполняемый файл**
  - имена с внешним связыванием доступны во всей программе. Подобные имена «экспортируются» из объектного файла, создаваемого компилятором.

- имена с внутренним связыванием доступны только в пределах файла, в котором они определены, но могут «разделяться» между всеми функциями этого файла.
- имена без связывания принадлежат одной функции и не могут разделяться вообще.
- время жизни, область видимости и связывание переменной зависят от места ее определения. По умолчанию

int i; // глобальная переменная

- Глобальное время жизни
- Файловая область видимости
- Внешнее связывание
- {
- int i;//локальная переменная
- ...
- Локальное время жизни
- Видимость в блоке
- Отсутствие связывания

- ощутимое значение явления связывания при работе с динамическими библиотеками, используется динамическое связывание функций, строится дерево зависимостей от so/dll файлов

## 23. Журналирование

Сохранение всех действий программы в лог-файл (Журнал)

Создаётся функция записи сообщения в журнал, и каждая функция при вызове её вызывает. Таким образом при отладке можно просмотреть каждое действие программы и понять как/из-за чего программа упала.

## 24. Классы памяти



**auto** - **автоматический** - локальные идентификаторы, память для которых выделяется при входе в блок, т.е. составной оператор, и освобождается при выходе из блока. Слово **auto** является сокращением слова **automatic**.

**static** - **статический** - локальные идентификаторы, существующие в процессе всех выполнений блока. В отличие от идентификаторов типа **auto**, для идентификаторов типа **static** память выделяется только один раз - в начале выполнения программы, и они существуют, пока программа выполняется.

**extern** - **внешний** - идентификаторы, называемые внешними, **external**, используются для связи между функциями, в том числе независимо скомпилированными функциями, которые могут находиться в различных файлах. Память, ассоциированная с этими идентификаторами, является постоянной, однако ее содержимое может меняться. Эти идентификаторы описываются вне функции.

**register** - **регистровый** - идентификаторы, подобные идентификаторам типа **auto**. Их значения, если это возможно, должны помещаться в регистрах машины для обеспечения быстрого доступа к данным.

Если класс памяти идентификатора не указан явно, то его класс памяти задается положением его определения в тексте программы. Если идентификатор определяется внутри функции, тогда его класс памяти **auto**, в остальных случаях идентификатор имеет класс памяти **extern**.

Предположим, что имеется программа на языке Си, исходный текст которой содержится в нескольких файлах. Для разделения данных (для связи) в функциях в этих файлах используются идентификаторы, определенные как **extern**. Если функция ссылается на внешний идентификатор, то файл, содержащий его, должен иметь описание или определение этого идентификатора. Явное задание класса памяти **extern** указывает на то, что этот идентификатор определен в другом файле, и здесь ему память не выделяется, а его описание дано лишь для проверки типа и для генерации кода.

Управлять временем жизни, областью видимости и связыванием переменной (до определенной степени) можно с помощью так называемых классов памяти.

В языке Си существует четыре класса памяти

- **auto**;
- **static**;
- **extern**;
- **register**.

### Auto:

Применим только к переменным, определенным в блоке.

```
int main(void)
{
 auto int i;
```

Переменная, принадлежащая к классу **auto**, имеет локальное время жизни, видимость в пределах блока, не имеет связывания.

По умолчанию любая переменная, объявленная в блоке или в заголовке функции, относится к классу автоматической памяти.

### Static:

Класс памяти static может использоваться с любыми переменными независимо от места их расположения.

- Для переменной вне какого-либо блока, static изменяет связывание этой переменной на внутреннее.
- Для переменной в блоке, static изменяет время жизни с автоматического на глобальное.

Статическая переменная, определенная вне какого-либо блока, имеет *глобальное время жизни, область видимости в пределах файла и внутреннее связывание*.

```
static int i;
void f1(void)
{
 i = 1;
}
void f2(void)
{
 i = 5;
}
```

Этот класс памяти скрывает переменную в файле, в котором она определена.

Статическая переменная, определенная в блоке, имеет *глобальное время жизни, область видимости в пределах блока и отсутствие связывания*.

```
void f(void)
{
 static int j;
 ...
}
```

- Такая переменная сохраняет свое значение после выхода из блока.
- Инициализируется только один раз.
- Если функция вызывается рекурсивно, это порождает новый набор локальных переменных, в то время как статическая переменная разделяется между всеми вызовами.

**Extern:**

Помогает разделить переменную между несколькими файлами.

|                                                                                                      |                                                                                                                          |
|------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------|
| <pre>// file_1.c int number;  // file_2.c int process(void) {     if (number &gt; 5)     ... }</pre> | <pre>// file_1.c int number;  // file_2.c extern int number;  int process(void) {     if (number &gt; 5)     ... }</pre> |
| error: 'number' undeclared                                                                           | OK                                                                                                                       |

Используется для переменных определенных как в блоке, так и вне блока.

`extern int i;`

Глобальное время жизни, файловая область видимости, связывание непонятное

```
{
 extern int i;
 ...
}
```

Глобальное время жизни, видимость в блоке, связывание непонятное

- Объявлений (`extern int number;`) может быть сколько угодно.
- Определение (`int number;`) должно быть только одно.
- Объявления и определение должны быть одинакового типа.

Замечание.

`extern int number = 5; // определение`

## Register:

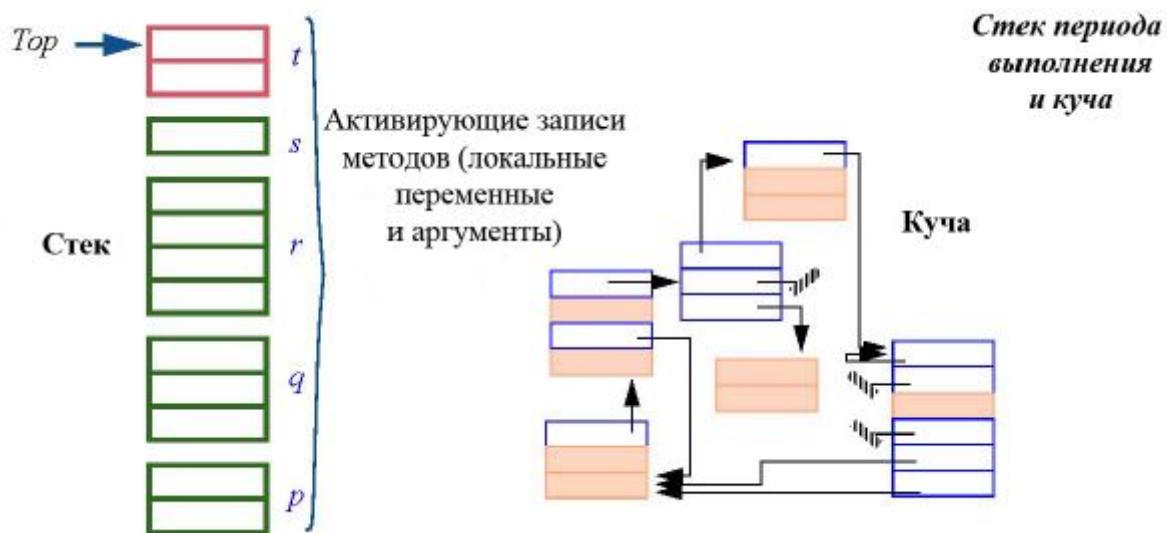
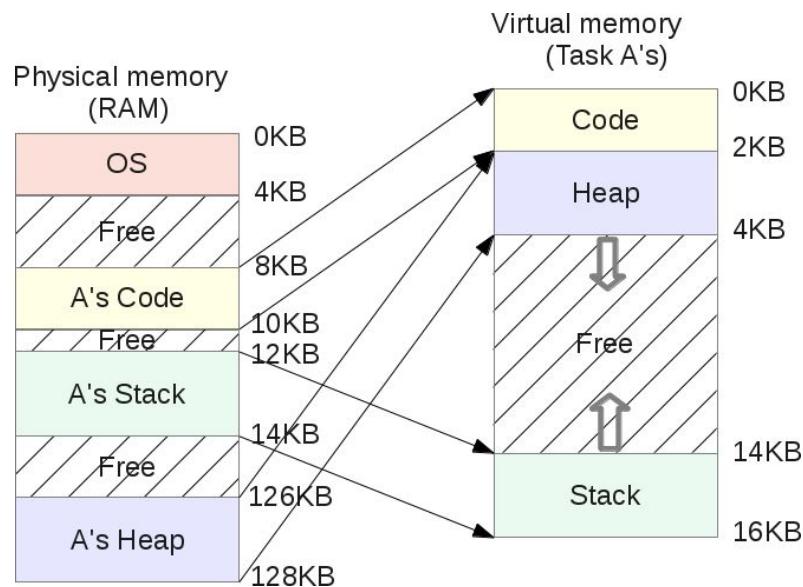
Использование класса памяти `register` – просьба (!) к компилятору разместить переменную не в памяти, а в регистре процессора.

- Используется только для переменных, определенных в блоке.
- Задает локальное время жизни, видимость в блоке и отсутствие связывания.
- Обычно не используется.

К переменным с классом памяти `register` нельзя применять операцию получения адреса `&`.

## 25. Стек и куча.

План ответа:



### - Стек

- Стек — это область оперативной памяти, которая создаётся для каждого потока. Он работает в порядке LIFO (Last In, First Out), то есть последний добавленный в стек кусок памяти будет первым в очереди на вывод из стека. Каждый раз, когда функция объявляет новую переменную, она добавляется в стек, а когда эта переменная пропадает из области видимости (например, когда функция заканчивается), она автоматически удаляется из стека. Когда стековая переменная освобождается, эта область памяти становится доступной для других стековых переменных.

### - Куча

- Куча — это хранилище памяти, также расположено в ОЗУ, которое допускает динамическое выделение памяти и не работает по принципу стека: это просто склад для ваших переменных. Когда вы выделяете в куче участок памяти для хранения переменной, к ней можно обратиться не только в потоке, но и во всем приложении. Именно так определяются глобальные переменные. По завершении приложения все выделенные участки памяти освобождаются. Размер кучи задается при запуске приложения, но, в отличие от стека, он ограничен лишь физически, и это позволяет создавать динамические переменные. В сравнении со стеком, куча работает медленнее, поскольку переменные разбросаны по памяти, а не сидят на верхушке стека. Некорректное управление памятью в куче приводит к замедлению её работы; тем не менее, это не уменьшает её важности — если вам нужно работать с динамическими или глобальными переменными, пользуйтесь кучей.
- **автоматическая память: использование и реализация**
  - auto — автоматическая (локальная). Автоматические переменные создаются при входе в функцию и уничтожаются при выходе из неё. Они видны только внутри функции или блока, в которых определены.
- **использование аппаратного стека (вызов функции, возврат управления из функции, передача параметров, локальные переменные, кадр стека)**
  - Аппаратный стек реализуется на базе оперативной памяти. Элементы стека расположены в оперативной памяти, каждый из них занимает одно слово. Регистр SP в любой момент времени хранит адрес элемента в вершине стека. Стек растет в сторону уменьшения адресов: элемент, расположенный непосредственно под вершиной стека, имеет адрес SP + 4 (при условии, что размер слова равен четырем байтам), следующий SP + 8 и т.д.
  - Одно из главных назначений аппаратного стека — поддержка вызовов подпрограмм. При вызове подпрограммы надо сохранить адрес возврата, чтобы подпрограмма могла по окончанию своей работы вернуть управление вызвавшей ее программе. В старых архитектурах, в которых аппаратный стек отсутствовал (например, в компьютерах IBM 360/370), точки возврата сохранялись в фиксированных ячейках памяти для каждой подпрограммы. Это делало невозможной рекурсию, т.е. повторный вызов той же подпрограммы непосредственно из ее текста или через цепочку промежуточных вызовов, поскольку при повторном вызове старое содержимое ячейки, хранящей адрес возврата, терялось. Во всех современных архитектурах точка возврата сохраняется в аппаратном стеке, что делает возможным рекурсию, а также параллельное выполнение нескольких легковесных процессов (нитей). Для вызова подпрограммы f служит команда call, которая осуществляет переход к подпрограмме f (т.е.

присваивает регистру PC адрес f ) и одновременно помещает старое содержимое регистра PC в стек:

- `call f ~ push PC;`  
`PC:= f;`

- В момент выполнения любой команды регистр PC содержит адрес следующей команды, т.е. фактически адрес возврата из подпрограммы f. Таким образом, команда call сохраняет в стеке точку возврата и осуществляет переход к подпрограмме f. Для возврата из подпрограммы используется команда return. Она извлекает из стека адрес возврата и помещает его в регистр PC:
  - `return ~ pop PC;`
- Как только мы вызываем функцию, в стеке для нее создается стековый кадр. Стековый кадр содержит локальные переменные, а также аргументы, которые были переданы вызывающей функцией. Помимо этого кадр содержит служебную информацию, которая используется вызванной функцией, чтобы в нужный момент возвратить управление вызвавшей функции. Точное содержание стека и схема его размещения в памяти могут быть разными в зависимости от процессорной архитектуры и используемой конвенции вызова. В данной статье мы рассматриваем стек на архитектуре x86 с конвенцией вызова, принятой в языке C (cdecl). На рисунке вверху изображен стековый кадр, разместившийся у верхушки стека.
- **динамическая память: использование и реализация**
  - Было ниже или выше
- **идеи реализации функций динамического выделения и освобождения память**
  - Вообще нет смысла расписывать, уже было выше или ниже все

---

## 26. Функции с переменным числом параметров.

В языке С есть возможность объявлять и использовать функции с переменным числом аргументов. Возможность эта обеспечивается особенностями работы со стеком при вызове функций: по умолчанию параметры передаются функции через стек, поэтому, технически, нет ограничения на количество передаваемых параметров - передавать можно сколько угодно. Остается лишь решить, как функция будет разбирать параметры. Пример решения данного вопроса будет представлен ниже.

Функции с переменным числом параметров объявляются как обычные функции, но вместо недостающих аргументов ставится многоточие (...). Тривиальным примером функции с переменным числом параметров может служить printf().

Собственно, для решения выше поставленного вопроса можно использовать стандартную библиотеку `<stdarg.h>`. Макросы из данной библиотеки (`va_arg`, `va_start`, `va_end`) используются для того, чтобы осуществить передачу функции переменного числа параметров. Функция должна иметь один или более известных параметров, которые следуют перед списком переменных параметров. Самый правый известный параметр (назовем его `last`) используется в качестве второго параметра в вызове `va_start()`. Прежде чем осуществлять доступ к одному из переменных параметров, должен быть инициализирован указатель (`argptr`), для чего используется вызов `va_start()`. После, параметры возвращаются с помощью `va_arg()` (с параметром `type` - типом следующего параметра).

По окончании работы с переменным числом параметров, для восстановления стека вызывается `va_end()` (иначе возникает аварийная ситуация).

Пример реализации:

```
#include <stdio.h>
#include <stdarg.h> // то, что нам и нужно
#define OK 0
```

```
/*
```

К примеру, мы хотим реализовать функцию, которая вернет сумму последовательности чисел. Первым аргументом будет число параметров, переданных функции.

```
*/
```

```
float our_function(int n, ...)
{
 float sum = 0, tmp;

 va_list argptr;

 va_start(argptr, n); // Инициализируем указатель

 for (; n; n--) // Можно также использовать while (n--)
 {
 tmp = va_arg(argptr, float);
 sum += tmp;
 }

 va_end(argptr);
```

```

 return sum;
}

int main(void)
{
 float sum = our_function(3, 0.5, 0.25, 0.125);
 printf("\n%f\n", sum);
 return OK;
}

```

---

## 27. Структуры

План ответа:

Структура — одна или несколько переменных объединенных под разным именем

Определение структурного типа:

Раздельные определения типа и  
переменных

```

struct date
{
 int day;
 int month;
 int year;
};

struct date birthday;
struct date exam;

```

Совмещенные определения типа  
и переменных

```

struct date
{
 int day;
 int month;
 int year;
} birthday, exam;

```

Тег структуры — имя, располагающееся за ключевым словом struct

```

struct point
{
```

```
 int x;
 int y;
}
```

Тег может быть опущен (безымянный тип). Не распознается без ключевого слова struct

Поле структуры — перечисленные в теле переменные

```
struct point
{
 int x;
 int y;
}
```

С целью оптимизации доступа компилятор может располагать поля в памяти не одно за другим, а по адресам кратным, например, размеру поля. Чтобы явно указать компилятору что так делать не нужно, можно воспользоваться директивой pragma pack

```
struct
{
 char a;
 int b;
} c1;

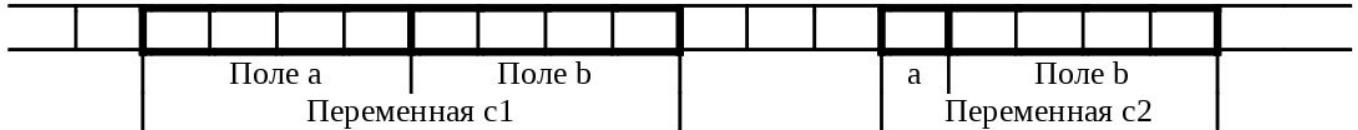
sizeof(c1) == 8

(char*) &c1 == &c1.a
```

```
#pragma pack(push, 1)
struct
{
 char a;
 int b;
} c2;
#pragma pack(pop)

sizeof(c2) == 5

(char*) &c2 == &c2.a
```



Для инициализации переменной структурного типа необходимо указать список значений, заключенный в фигурные скобки.

Значения в списке должны появляться в том же порядке, что и имена полей структуры.

```
struct date
{
 int day;
```

```
 int month
 int year;
};

...
struct date today = {11, 1, 2018};
```

Если значений меньше, чем полей структуры, оставшиеся поля инициализируются нулями.  
`struct date tomorrow = {12} // month = 0; year = 0;`

В C99 возможна инициализация по полям

```
struct date tomorrow = {.day = 12, .month = 1, .year = 2018};
```

Возможна комбинация со старым способом

```
struct date yesterday = {.day = 10, 1, .year = 2018};
```

Операции над структурами:

Доступ к полю структуры осуществляется с помощью операции «.», или «->» если доступ к структуре осуществляется по указателю

```
struct date today;
struct date *yesterday;
yesterday = malloc(sizeof(struct date));
if (!yesterday)
...
today.day = 11;
(*yesterday).day = 10;
yesterday->month = 1;
```

Структурные переменные одного типа можно присваивать друг другу

```
yesterday = today;
```

Структуры нельзя сравнивать с помощью «==» и «!=»

Структуры можно передавать в параметры функции и возвращать из неё в качестве значения

```
void print(struct date d)
{
 printf("%02d.%02d.%04d", d.day, d.month, d.year);
}
```

Передача структур с помощью указателей

Эффективность (экономия стека и времени на копировании данных).

Необходимость изменения переменной (например, FILE\*).

```

void print_ex(const struct date *d)
{
 printf("%02d.%02d.%04d", d->day, d->month, d->year);
}

struct date get_student_date(void)
{
 struct date d = {25, 1, 2017};
 return d;
}

```

Flexible array member

```
struct {int n, double d[]};
```

Структуры с полем неопределенного размера.

Подобное поле должно быть последним

Нельзя создавать массивы структур с таким полем

Подобные структуры не могут стоять «в середине» другой структуры

Операция sizeof не учитывает размер такого поля (За исключением выравнивания)

Если в массиве нет элементов, обращение к такому полю — неопределённое поведение.

**P.S.** (Для последующих курсов) Вот это вот всё <https://vk.com/id265762564> я написал. А ещё про журналирование и списки Беркли. Можете написать, поинтересоваться как у меня дела. Не вылетел ли ещё, как учёба идёт и всё такое. Долгов сейчас много, особенно по «Основам электроники». Удачи вам на экзамене.

## 28. Объединения.

План ответа:

**понятие «объединение»;**

Объединение, как и структура, содержит одно или несколько полей возможно разного типа. Однако все поля объединения разделяют одну и ту же область памяти.

```

union
{
 int i;
 double d;
} u;

```

## **определение переменной-объединения, способы инициализации переменной-Объединения;**

Присвоение значения одному члену объединения обычно изменит значение других членов.

```
printf("u.i %d, u.d %g\n", u.i, u.d);
// u.i 2293664, u.d 1.7926e-307
u.i = 5;
printf("u.i %d, u.d %g\n", u.i, u.d);
// u.i 5, u.d 1.79255e-307
u.d = 5.25;
printf("u.i %d, u.d %g\n", u.i, u.d);
// u.i 0, u.d 5.25
```

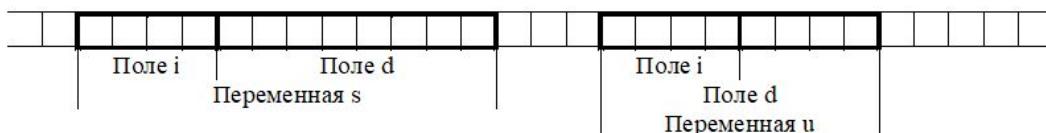
## **Инициализация**

```
union u_t
{
 int i;
 double d;
};

...
union u_t u_1 = {1};
// только с99
union u_t u_2 = { .d = 5.25 };
```

## **особенности выделения памяти под объединения.**

- Экономия места.



**использование объединений. !!! ЕБАТЬ ВАНЯ ТЫ ТРИВИАЛЬНЫЕ ПРИМЕРЫ  
ПРИВОДИШЬ !!!**

## !!!МАРСЕЛЬ Я ТЕБЯ В РОТ ЕБАЛ!!!

```
struct library_item {
 int number;
 int item_type;
 union {
 struct {
 char author[NAME_LEN + 1];
 char title[TITLE_LEN + 1];
 char publisher[PUBLISHER_LEN + 1];
 int year;
 } book;
 struct {
 char title[TITLE_LEN + 1];
 int year;
 int volume;
 } magazine;
 } item;
};
```

---

## 29. Динамический расширяемый массив.

План ответа:

функция realloc и особенности ее использования;

`void* realloc(void *ptr, size_t size);`

• `ptr == NULL && size != 0`

Выделение памяти (как malloc)

• `ptr != NULL && size == 0`

– Освобождение памяти аналогично `free()`. Результат можно (но не обязательно!) передать во `free()`.

• `ptr != NULL && size != 0`

Перевыделение памяти. В худшем случае:

• выделить новую область

• скопировать данные из старой области в новую

• освободить старую область

- описание типа;

```
struct dyn_array
{
 int len;
 int allocated;
 int step;
 int *data;
};

#define INIT_SIZE 1
void init_dyn_array(struct dyn_array *d)
{
 d->len = 0;
 d->allocated = 0;
 d->step = 2;
 d->data = NULL;
}
```

---

### 30. Линейный односвязный список.

План ответа:

- описание типа

#### Линейный односвязный список



#### Отличия списка от массива

- Размер массива фиксирован, а списка нет.
- Списки можно переформировывать, изменяя несколько указателей.
- При удалении или вставке нового элемента в список адрес остальных не меняется.

- основные операции

- добавление элемента в список
- поиск элемента в списке
- удаление элемента

- обработка всех элементов списка (печать, сортировка и тд)

Допустим, что у нас есть структура:

```
struct template
{
 Int data;
 Struct template *next;
};
```

Добавление элемента в список:

```
Struct template * add(struct template *list, int n)
{
 Struct template *tmp = malloc(sizeof(struct template));

 tmp->data = n;
 tmp->next = NULL;

 list->next = tmp;

 Return list;
};
```

Удаление элемента из списка (корня):

```
Struct template * delete(struct template *list)
{
 Struct template *tmp;

 Tmp = list->next;
 free(list);

 Return tmp;
}
```

Любая обработка целого списка происходит путем обхода всех его элементов:

```
Void example(struct template *list)
```

```
{
 Struct template *current = list;
 For (; current; current = current->next);
}
```

---

## 31. Двоичные деревья поиска.

**План ответа:**

**описание типа**

- *Дерево* - это связный ациклический граф.

- *Двоичным деревом поиска* называют дерево, все вершины которого упорядочены, каждая вершина имеет не более двух потомков (назовём их левым и правым), и все вершины, кроме корня, имеют родителя.

**основные операции**

- добавление элемента в дерево
- поиск в дереве
- обход дерева
- рекурсивный и нерекурсивный поиск

**рекурсивный и нерекурсивный поиск**

```
struct tree_node* lookup_1(struct tree_node *tree,
 const char *name)
{
 int cmp;

 if (tree == NULL)
 return NULL;

 cmp = strcmp(name, tree->name);
 if (cmp == 0)
 return tree;
 else if (cmp < 0)
 return lookup_1(tree->left, name);
 else
 return lookup_1(tree->right, name);
}
```

```

struct tree_node* lookup_2(struct tree_node *tree,
 const char *name)
{
 int cmp;

 while (tree != NULL)
 {
 cmp = strcmp(name, tree->name);
 if (cmp == 0)
 return tree;
 else if (cmp < 0)
 tree = tree->left;
 else
 tree = tree->right;
 }

 return NULL;
}

```

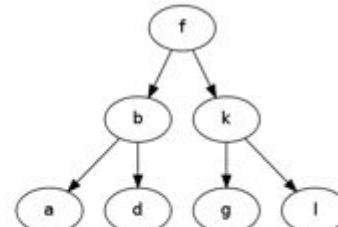
## язык DOT

- DOT - язык описания графов.
- Граф, описанный на языке DOT, обычно представляет собой текстовый файл с расширением .gv в понятном для человека и обрабатывающей программы формате.
- В графическом виде графы, описанные на языке DOT, представляются с помощью специальных программ, например Graphviz. («Введение в GraphViz»)

```

// Описание дерева на DOT // Оформление на странице Trac
digraph test_tree { {{{
f -> b; #!graphviz
f -> k; digraph test_tree {
b -> a; f -> b;
b -> d; f -> k;
k -> g; b -> a;
k -> l; b -> d;
} k -> g;
} k -> l;
}}}} }}}

```



[Edit this page](#) | [Attach file](#) | [Rename page](#)

```

void to_dot(struct tree_node *tree, void *param)
{
 FILE *f = param;

 if (tree->left)
 fprintf(f, "%s -> %s;\n", tree->name, tree->left->name);

 if (tree->right)
 fprintf(f, "%s -> %s;\n", tree->name, tree->right->name);
}

void export_to_dot(FILE *f, const char *tree_name,
 struct tree_node *tree)
{
 fprintf(f, "digraph %s {\n", tree_name);
 apply_pre(tree, to_dot, f);
 fprintf(f, "}\n");
}

```

31

## 32. Директивы препроцессора, макросы.

План ответа:

**классификация директив препроцессора;**

- Макроопределения  
#define, #undef
- Директива включения файлов  
#include
- Директивы условной компиляции  
#if, #ifdef, #endif и др.

Остальные директивы (#pragma, #error, #line и др.) используются реже.

**правила, справедливые для всех директив препроцессора;**

- Директивы всегда начинаются с символа "#".
- Любое количество пробельных символов может разделять лексемы в директиве.
- Директива заканчивается на символе '\n'.
- Директивы могут появляться в любом месте программы
- Любое количество пробельных символов могут разделять лексемы в директиве.

# define N 1000

- Директива заканчивается на символе '\n'.

```
#define DISK_CAPACITY (SIDES *
 TRACKS_PER_SIDE * \
 SECTORS_PER_TRACK * \
 BYTES_PER_SECTOR)
```

## Простые макросы

#define идентификатор список-замены

```
#define PI 3.14
#define EOS '\0'
#define MEM_ERR "Memory allocation error."
```

Используются:

- В качестве имен для числовых, символьных и строковых констант.
- Незначительного изменения синтаксиса языка.

```
#define BEGIN {
#define END }
#define INF_LOOP for(; ;)
```

- Переименования типов.

```
#define BOOL int
```

- Управления условной компиляцией.

## Макросы с параметрами

#define идентификатор(x1, x2, ..., xn) список-замены

- Не должно быть пробела между именем макроса и (.
- Список параметров может быть пустым.

```
#define MAX(x, y) ((x) > (y) ? (x) : (y))
#define IS_EVEN(x) ((x) % 2 == 0)
```

Где-то в программе

```
i = MAX(j + k, m - n);
```

```

// i = ((j + k) > (m - n) ? (j + k) : (m - n));

if (IS_EVEN(i))
// if (((i) % 2 == 0))
 i++;

```

## Макросы с переменным числом параметров (C99)

```

#ifndef NDEBUG
#define DBG_PRINT(s, ...) printf(s, __VA_ARGS__)
#else
#define DBG_PRINT(s, ...) ((void) 0)
#endif

```

## Общие свойства макросов

- Список-замены макроса может содержать другие макросы.
- Препроцессор заменяет только целые лексемы, не их части.
- Определение макроса остается «известным» до конца файла, в котором этот макрос объявляется.
- Макрос не может быть объявлен дважды, если эти объявление не тождественны.
- Макрос может быть «разопределен» с помощью директивы #undef.

## Сравнение макросов с параметрами и функций

### *Преимущества*

- программа может работать немного быстрее;
- макросы "универсальны".

### Недостатки

- скомпилированный код становится больше;
- n = MAX(i, MAX(j, k));
- типы аргументов не проверяются;
- нельзя объявить указатель на макрос;
- макрос может вычислять аргументы несколько раз.
- n = MAX(i++, j);

## Скобки в макросах

- Если список-замены содержит операции, он должен быть заключен в скобки.
- Если у макроса есть параметры, они должны быть заключены в скобки в списке-замены.

```
#define TWO_PI 2 * 3.14
```

```

f = 360.0 / TWO_PI;
// f = 360.0 / 2 * 3.14;

#define SCALE(x) (x * 10)

j = SCALE(i + 1);
// j = (i + 1 * 10);

```

### **Создание длинных макросов**

```

// 1
#define ECHO(s){gets(s); puts(s);}

if (echo_flag)
 ECHO(str);
else
 gets(str);

// 2
#define ECHO(s) (gets(s), puts(s))

ECHO(str);

#define ECHO(s)
do {
 gets(s);
 puts(s);
}
while(0);

```

### **Переопределенные макросы**

- \_\_LINE\_\_ - номер текущей строки (десятичная константа)
- \_\_FILE\_\_ - имя компилируемого файла
- \_\_DATE\_\_ - дата компиляции
- \_\_TIME\_\_ - время компиляции
  
- и др.

Эти идентификаторы нельзя переопределять или отменять директивой `undef`.

- \_\_func\_\_ - имя функции как строка (GCC only, C99 и не макрос)
- 

### 33. Директивы препроцессора, условная компиляция, операции # и ##.

План ответа:

- **классификация директив препроцессора**
  - макроопределения `#define`, `#undef`
  - директива включения файлов `#include`
  - директива условной компиляции `#if`, `#ifdef <...>`
  - Остальные
    - `pragma`
    - `error`
    - `line`
- **правила, справедливые для всех директив препроцессора**
  - директивы всегда начинаются с символа "#"
  - любое количество пробельных символов может разделять лексемы в директиве
  - директива заканчивается на символе '\n'
  - директивы могут появляться в любом месте программы
  - любое количество пробельных символов могут разделять лексемы в директиве
  - директива заканчивается на символе '\n'
- **директива error**

`#error` сообщение

```
•
#if defined(OS_WIN)
...
#elif defined(OS_LIN)
...
#elif defined(OS_MAC)
...
#else
#error Unsupported OS!
#endif
```

- **операция “#”**

«Операция» # конвертирует аргумент макроса в строковый литерал.

```
#define PRINT_INT(n) printf(#n " = %d\n", n)
•
#define TEST(condition, ...) ((condition) ? \
 printf("Passed test %s\n", #condition) : \
 printf(__VA_ARGS__))
```

Где-то в программе

```
PRINT_INT(i / j);
// printf("i/j" " = %d", i/j);
TEST(voltage <= max_voltage,
 "Voltage %d exceed %d", voltage, max_voltage);
```

#### - **операция “##”**

«Операция» ## объединяет две лексемы в одну.

```
•
#define MK_ID(n) i##n
•
```

Где-то в программе

```
•
int MK_ID(1), MK_ID(2);
// int i1, i2;
•
```

Более содержательный пример

```
•
#define GENERAL_MAX(type)
type type##_max(type x, type y)
{
 return x > y ? x : y;
}
```

#### - **директива pragma (once, pack)**

Директива #pragma позволяет добиться от компилятора специфичного поведения.

---

34. Inline-функции.  В разделе “Вопросы к семинарам”

---

35. Списки из ядра операционной системы Linux (списки Беркли).

Список Беркли — встраиваемый циклический двусвязный список, в основе которого лежит структура

```
struct list_head
{
 struct list_head *next, *prev;
};
```

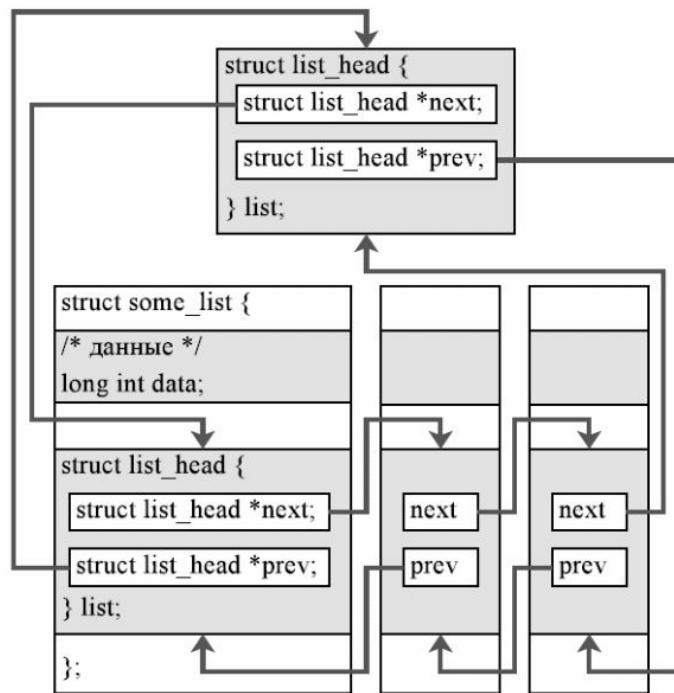
В отличие от обычных списков, где данные содержатся в элементах списка, структура list\_head должна быть частью самих данных

```
struct data
{
 int i;
 struct list_head list;
 ...
};
```

Структуру struct list\_head можно поместить в любом месте в определении структуры.  
struct list\_head может иметь любое имя.

В структуре может быть несколько полей типа struct list\_head.

# Списки в стиле Беркли



Для обращения к списку нужно посчитать количество бит до структуры list\_head  
int offset = (int) (&((struct s\*) 0)->i)

Код функций и макросов:

```
static inline void list_add(struct list_head *new,
 struct list_head *head)
{
 __list_add(new, head, head->next);
}

static inline void list_add_tail(struct list_head *new,
 struct list_head *head)
{
 __list_add(new, head->prev, head);
}
```

```

#define list_for_each(pos, head) \
 for (pos = (head)->next; pos != (head); pos = pos->next)

#define list_for_each_prev(pos, head) \
 for (pos = (head)->prev; pos != (head); pos = pos->prev)

#define list_for_each_entry(pos, head, member) \
 for (pos = list_entry((head)->next, typeof(*pos), member); \
 &pos->member != (head); \
 pos = list_entry(pos->member.next, typeof(*pos), member))

#define list_for_each_safe(pos, n, head) \
 for (pos = (head)->next, n = pos->next; pos != (head); \
 pos = n, n = pos->next)

#define list_entry(ptr, type, member) \
 container_of(ptr, type, member)

#define container_of(ptr, type, field_name) (\
 (type *) ((char *) (ptr) - offsetof(type, field_name)))

#define offsetof(TYPE, MEMBER) \
 ((size_t) &((TYPE *)0)->MEMBER)

```

## 36. Битовые операции. Битовые поля.

План ответа:

### **Проверка битов(&)**

```

unsigned char a = 0x46; // 01000110b
unsigned char b = 0x44; // 01000100b
unsigned char mask = 0x06; // 000000110b
printf("a & mask %x, res %d\n", a & mask, (a & mask) == mask);
printf("b & mask %x, res %d\n", b & mask, (b & mask) == mask);

```

### **Обнуление битов (&=)**

```

unsigned char a = 0x46; // 01000110b
unsigned char mask_1 = 0xbf; // 10111111b

```

```

unsigned char mask_2 = 0xf9; // 11111001b
printf("a & mask_1 %x\n", a & mask_1);
printf("a & mask_2 %x\n", a & mask_2);

```

### **Установка битов (&)**

```

unsigned char a = 0x40; // 01000000b
unsigned char mask_1 = 0x06; // 00000110b
unsigned char mask_2 = 0x44; // 01000100b
printf("a | mask_1 %x\n", a | mask_1);
printf("a | mask_2 %x\n", a | mask_2);

```

### **Смена значений битов (^)**

```

unsigned char a = 0x46; // 01000110b
unsigned char mask_1 = 0x44; // 01000100b
unsigned char mask_2 = 0xFF; // 11111111b
printf("a ^ mask_1 %x\n", a ^ mask_1);
printf("a ^ mask_2 %x\n", a ^ mask_2);

```

### **Сдвиг вправо (>>)**

```

unsigned char a = 0xFF; // 11111111b
printf("a >> 1 = %2x\n", a >> 1);
printf("a >> 4 = %2x\n", a >> 4);

```

### **Сдвиг влево (<<)**

```

unsigned char a = 0x01; // 00000001b
printf("a << 1 = %2x\n", a << 1);
printf("a << 4 = %2x\n", a << 4);

```

| Операция         | Название                    | Нотация             | Класс      | Приоритет | Ассоциат.     |
|------------------|-----------------------------|---------------------|------------|-----------|---------------|
| <b>~ (унар.)</b> | Побитовое «НЕ»              | <b>~x</b>           | Префиксные | 15        | Справа налево |
| <b>&lt;&lt;</b>  | Сдвиг влево                 | <b>x &lt;&lt; y</b> | Инфиксные  | 11        | Слева направо |
| <b>&gt;&gt;</b>  | Сдвиг вправо                | <b>x &gt;&gt; y</b> |            |           |               |
| <b>&amp;</b>     | Побитовое «И»               | <b>x &amp; y</b>    | Инфиксные  | 8         | Слева направо |
| <b>^</b>         | Побитовое исключающее «ИЛИ» | <b>x ^ y</b>        | Инфиксные  | 7         | Слева направо |
| <b> </b>         | Побитовое «ИЛИ»             | <b>x   y</b>        | Инфиксные  | 6         | Слева направо |

|                        |                                            |                            |                |               |
|------------------------|--------------------------------------------|----------------------------|----------------|---------------|
| <code>&lt;&lt;=</code> | Присваивание со сдвигом влево              | <code>x &lt;&lt;= y</code> | Инфиксные<br>2 | Справа налево |
| <code>&gt;&gt;=</code> | Присваивание со сдвигом вправо             | <code>x &gt;&gt;= y</code> |                |               |
| <code>&amp;=</code>    | Присваивание с побитовым «И»               | <code>x &amp;= y</code>    |                |               |
| <code>^=</code>        | Присваивание с побитовым исключающим «ИЛИ» | <code>x ^= y</code>        |                |               |
| <code> =</code>        | Присваивание с побитовым «ИЛИ»             | <code>x  = y</code>        |                |               |

различие между битовыми и логическими операциями;

```
#include <stdio.h>

int main(void)
{
 unsigned char a = 0x01;
 unsigned char b = 0x02;

 if (a & b)
 printf("a & b true\n");
 else
 printf("a & b false\n");

 if ((b & 1) == 1)
 printf("odd\n");
 else
 printf("even\n");

 a = 0;
 if (a & b / a) // !!!
 printf("true\n");
 else
 printf("false\n");

 return 0;
}
```

```
#include <stdio.h>

int main(void)
{
 unsigned char a = 0x01;
 unsigned char b = 0x02;

 if (a && b)
 printf("a && b true\n");
 else
 printf("a && b false\n");

 if ((b && 1) == 1)
 printf("odd\n");
 else
 printf("even\n");

 a = 0;
 if (a && b / a)
 printf("true\n");
 else
 printf("false\n");

 return 0;
}
```

**битовые поля: описание, использование, ограничения использования.**

*Битовое поле* - особый тип структуры, определяющей, какую длину имеет каждый член в битах.

Стандартный вид объявления битовых полей следующий:

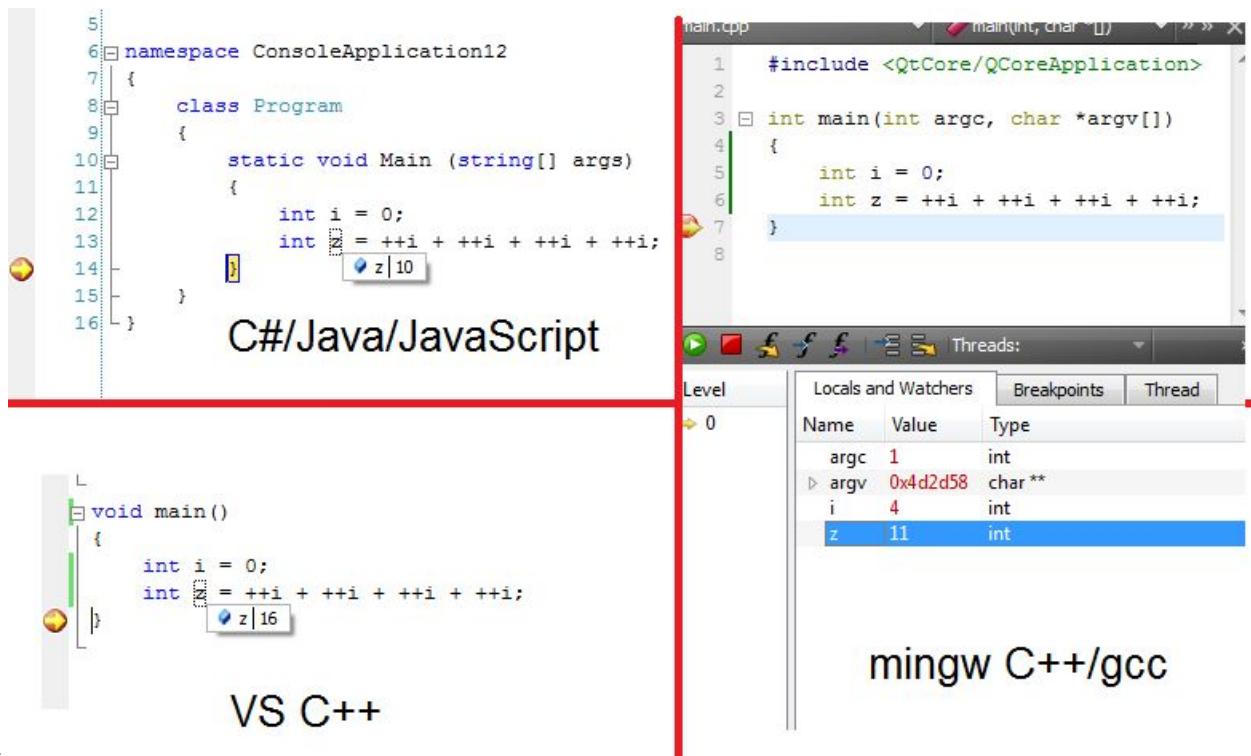
```

struct имя_структурьи
{
 тип имя1: длина;
 тип имя2: длина;
 ...
 тип имяN: длина;
};

```

Битовые поля должны объявляться как целые, unsigned или signed.

### 37. Неопределенное поведение. (отчасти написано ниже)



План ответа:

- **особенности вычисления выражений с побочным эффектом**
- **понятие «точка следования»**
  - Точки следования (sequence points) - это некие точки в программе, где состояние реальной программы полностью соответствует состоянию абстрактной машины, описанной в Стандарте. С помощью точек следования стандарт объясняет, что может, а чего не может делать компилятор и что нам нужно сделать, чтобы написать корректный код. В каждой точке следования все побочные эффекты кода, который уже выполнен, уже случились, а побочные эффекты для кода, который еще не был

выполнен, еще не случились. Не вдаваясь в детали: запись значения в переменную при присваивании есть пример побочного эффекта.

## - расположение «точек следования»

### Что такое точка следования

Всегда хочется, чтобы компилятор сгенерировал быстрый маленький и наиболее эффективный код из имеющегося исходника, разумеется, чтобы при этом все работало корректно. Иногда для этого компилятору требуется изменить порядок выполнения кода относительно того, как он был написан изначально и, возможно, провести другую оптимизацию.

Точки следования (sequence points) - это некие точки в программе, где состояние реальной программы полностью соответствует состоянию абстрактной машины, описанной в Стандарте. С помощью точек следования стандарт объясняет, что может, а чего не может делать компилятор и что нам нужно сделать, чтобы написать корректный код.

### Где находятся точки следования

1. В конце каждого полного выражения(Глава Стандарта 1.9/16). Обычно они помечены точкой с запятой ;
2. В точке вызова функции (1.9/17). Но после вычисления всех аргументов. Это и для inline функций в том числе.
3. При возвращении из функции. (1.9/17) Есть точка следования сразу после возврата функции, перед тем как любой другой код из вызвавшей функции начал выполняться.
4. (1.9/18) После первого выражения (здесь оно называется 'a') в следующих конструкциях:

a || b

a && b

a , b

a ? b : c

Вычисление здесь идет слева направо. То есть левое выражение (по имени 'a') вычисляется и все побочные эффекты от такого вычисления завершаются. Потом, если все значение всего выражения известно, правое выражение не вычисляется, иначе вычисляется.

Правило слева-направо не работает для переопределенных операторов. В этом случае переопределенный оператор ведет себя как обычная функция. Еще такой момент - ?: не может быть переопределен по стандарту.

(Упомянутая запятая это оператор запятая. Она не имеет никакого отношения к запятой, разделяющей аргументы функции.)

### Примеры:

```
i = ++i; // undefined behavior, переменная модифицируется дважды
i = i + 1; // все в порядке
i ? i=1 : i=5; // все в порядке (там, где знак ? есть точка следования, а потом выполнится
лишь одно из выражений)
i=1; i++; // все в порядке (после каждого выражения находится точка следования)
i=1, i++; // все в порядке (на операторе запятая находится точка следования)
```

### Пример с функциями:

```
void f(int, int);
int g();
int h();
f(g(), h());
```

По Стандарту неизвестно, какая из функций `g` или `h` будет вызвана первой, но известно, что `f()` будет вызвана последней.

Разработчикам gcc периодически сабмитят баги по этому поводу, в итоге в Frequently Reported Bugs in GCC было внесено:

```
Результат следующих операций непредсказуем и это полностью соответствует
стандарту
x[i]=++i
foo(i,++i)
i*(++i) /* special case with foo=="operator*" */
std::cout << i << ++i /* foo(foo(std::cout,i),++i) */
```

Вот пример такой баги: Bug13403. Их таких там очень много.

В следующей программе

```
#include <iostream>
int main(){
 int i=0;
 i = 6+ i++ + 2000;
 std::cout << i << std::endl;
 return 0;
}
```

Результат единица! А должен быть 2006.

Если убрать "`i++`", то результат правильный ("2006"). Но пока есть "`i++`" внутри выражения, я могу делить, умножать, вычитать, складывать, все равно результат всегда "1".

---

## 38. Библиотеки.

### Библиотеки

Библиотека включает в себя

- заголовочный файл;
- откомпилированный файл самой библиотеки;
- библиотеки меняются редко – нет причин перекомпилировать каждый раз;
- двоичный код предотвращает доступ к исходному коду.

Библиотеки делятся на

- статические;
- динамические.

### **Статические библиотеки**

Связываются с программой в момент компоновки. Код библиотеки помещается в исполняемый файл.

«+»

Исполняемый файл включает в себя все необходимое.

Не возникает проблем с использованием не той версии библиотеки.

«-»

«Размер».

При обновлении библиотеки программу нужно пересобрать.

### **Динамические библиотеки**

Подпрограммы из библиотеки загружаются в приложение во время выполнения. Код библиотеки не помещается в исполняемый файл.

«+»

Несколько программ могут «разделять» одну библиотеку.

Меньший размер приложения (по сравнению с приложением со статической библиотекой).

Средство реализации плагинов.

Модернизация библиотеки не требует перекомпиляции программы.

Могут использовать программы на разных языках.

«-»

Требуется наличие библиотеки на компьютере.

Версионность библиотек.

Способы использования динамических библиотек

динамическая компоновка;

динамическая загрузка.

### **Использование статической библиотеки**

Сборка библиотеки

- Компиляция  
`gcc -std=c99 -Wall -Werror -c arr_lib.c`
- Упаковка  
`ar rc libarr.a arr_lib.o`

- Индексирование  
`ranlib libarr.a`
- Сборка приложения  
`gcc -std=c99 -Wall -Werror main.c libarr.a -o test.exe`  
 Или  
`gcc -std=c99 -Wall -Werror main.c -L. -larr -o test.exe`

### **Использование динамической библиотеки (динамическая компоновка)**

Сборка библиотеки

- Компиляция  
`gcc -std=c99 -Wall -Werror -c arr_lib.c`
- Компоновка  
`gcc -shared arr_lib.o -Wl,--subsystem,windows -o arr.dll`
- Сборка приложения  
`gcc -std=c99 -Wall -Werror -c main.c`  
`gcc main.o -L. -larr -o test.exe`

### **Использование динамической библиотеки (динамическая загрузка)**

Сборка библиотеки

- Компиляция  
`gcc -std=c99 -Wall -Werror -c arr_lib.c`
- Компоновка  
`gcc -shared arr_lib.o -Wl,--subsystem,windows -o arr.dll`
- Сборка приложения  
`gcc -std=c99 -Wall -Werror main.c -o test.exe`

## 39. Абстрактный тип данных

План ответа:

**Понятие «модуль», преимущества модульной организации программы.**

- Программу удобно рассматривать как набор независимых модулей.
- Модуль состоит из двух частей: интерфейса и реализации.
- *Интерфейс* описывает, что модуль делает. Он определяет идентификаторы, типы и подпрограммы, которые будут доступны коду, использующему этот модуль.

- Реализация описывает, как модуль выполняет то, что предлагает интерфейс.
- У модуля есть один интерфейс, но реализаций, удовлетворяющих этому интерфейсу, может быть несколько.
- Часть кода, которая использует модуль, называют *клиентом*.
- Клиент должен зависеть только от интерфейса, но не от деталей его реализации.

## Преимущества

- Абстракция (как средство борьбы со сложностью)
 

Когда интерфейсы модулей согласованы, ответственность за реализацию каждого модуля делегируется определенному разработчику.
- Повторное использование
 

Модуль может быть использован в другой программе.
- Сопровождение
 

Можно заменить реализацию любого модуля, например, для улучшения производительности или переноса программы на другую платформу.

## Разновидности модулей.

### • Набор данных

Набор связанных переменных и/или констант. В Си модули этого типа часто представляются только заголовочным файлом. (float.h, limits.h.)

### • Библиотека

Набор связанных функций.

### • Абстрактный объект

Набор функций, который обрабатывает скрытые данные.

### • Абстрактный тип данных

Абстрактный тип данных – это интерфейс, который определяет тип данных и операции над этим типом. Тип данных называется абстрактным, потому что интерфейс скрывает детали его представления и реализации.

## Организация модуля в языке Си. Неполный тип в языке Си.

- В языке Си интерфейс описывается в заголовочном файле (\*.h).
- В заголовочном файле описываются макросы, типы, переменные и функции, которые клиент может использовать.
- Клиент импортирует интерфейс с помощью директивы препроцессора include.
- Реализация интерфейса в языке Си представляется одним или несколькими файлами с расширением \*.c.
- Реализация определяет переменные и функции, необходимые для обеспечения возможностей, описанных в интерфейсе.

- Реализация обязательно должна включать файл описания интерфейса, чтобы гарантировать согласованность интерфейса и реализации.

### **Неполный тип в языке Си**

• Стандарт Си описывает неполные типы как «типы которые описывают объект, но не предоставляют информацию нужную для определения его размера».

```
struct t;
```

- Пока тип неполный его использование ограничено.

- Описание неполного типа должно быть закончено где-то в программе.

- Допустимо определять указатель на неполный тип

```
typedef struct t *T;
```

- Можно

- определять переменные типа T;

- передавать эти переменные как аргументы в функцию.

- Нельзя

- применять операцию обращения к полю (->);

- разыменовывать переменные типа T.

## **Общие вопросы проектирования абстрактного типа данных.**

- Абстрактный тип данных

Абстрактный тип данных – это интерфейс, который определяет тип данных и операции над этим типом. Тип данных называется абстрактным, потому что интерфейс скрывает детали его представления и реализации.

### **Трудности, улучшения и т.п.**

- Именование

В примерах использовались имена функций, которые подходят для многих АТД (create, destroy, is\_empty). Если в программе будет использоваться несколько разных АТД, это может привести к конфликту. Поэтому имеет смысл добавлять название АТД в название функций (stack\_create, stack\_destroy, stack\_is\_empty).

- Обработка ошибок

- Интерфейс это своего рода контракт.

- Интерфейс обычно описывает *проверяемые* ошибки времени выполнения и *непроверяемые* ошибки времени выполнения и исключения.

- Реализация не гарантирует обнаружение непроверяемых ошибок времени выполнения. Хороший интерфейс избегает таких ошибок, но должен описать их.

–Реализация гарантирует обнаружение проверяемых ошибок времени выполнения и информирование клиентского кода.

- «Общий» АТД

- Хотелось бы чтобы стек мог «принимать» данные любого типа без модификации файла stack.h.

- Программа не может создать два стека с данными разного типа.

- 

- Решение – использовать void\* как тип элемента, НО:

- элементами могут быть динамически выделяемые объекты, но не данные базовых типов int, double;

- стек может содержать указатели на что угодно, очень сложно гарантировать правильность.

---

## ----- Вопросы к семинарам -----

План ответа:

- ❑ Доступ к переменной (чтение, запись) к переменным с модификатором const
  - ❑ ПРАВИЛО: провести линию по звездочке, все, что справа - относится к переменной, все, что слева - к типу, на который она указывает
    - ❑ int \*const p1
      - ❑ Справа находится p1, и это p1 константа. Тип, на который p1 указывает, это int. Значит получился константный указатель на int. Его можно инициализировать лишь однажды и больше менять нельзя.  
p1 = NULL; // ERROR  
\*p1 = 3; // OK
    - ❑ int const\* p2
      - ❑ Указатель на константную память! Нельзя менять значение по указателю, но можно менять указатель.
      - ❑ int q = 1;
      - ❑ const int \*p;
      - ❑ p = &q; // На что указывает p можно менять
      - ❑ \*p = 5; // Ошибка, число менять уже нельзя
    - ❑ const int\* p3

- Аналогично + описано подробнее выше
- Передача массива по значению в функцию, размещение массивов в памяти (куча, стек)

### □ STACK

- `int a[5];`
- `int a[5];`
- `int a[] = {1, 2, 3};`

### □ HEAP

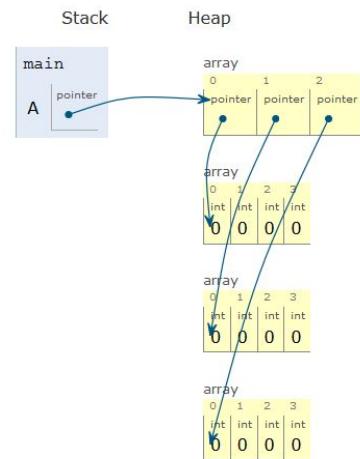
- Матрица как массив указателей

```
int** allocate_matrix(int n, int m)
{
 int **data = calloc(n, sizeof(int*));

 for (int i = 0; i < n; i++)
 data[i] = malloc(m * sizeof(int)); //calloc

 return data;
}

int main(void)
{
 int** A = allocate_matrix(5, 5);
}
```



## □ Метод alloca, использование, ограничения.

- `void* alloca(size_t size)`
- Эта функция не определена стандартом ANSI C. Функция alloca() выделяет size байт памяти из СТЕКА системы (не из кучи) и возвращает на него указатель. Если запрос на выделение памяти не выполнен, то возвращается нулевой указатель.
- **Выделенная с использованием alloca() память автоматически освобождается при выходе из функции, которая вызвала alloca(). Это означает, что указатель, возвращенный функцией alloca(), никогда не служит аргументом функции free().**
- **ЗАМЕТКА: По техническим причинам, чтобы гарантировать сохранность стека, всякая функция, в которой используется вызов alloca(), должна содержать хотя бы одну локальную переменную, которой присваивается значение.**

## □ Граф, представление графа в виде двумерного массива.

- Матрица A(NxN), где N-количество вершин графа
- $a[i][j] \in [0, 1]$
- 1, если из  $i$ -той вершины в  $j$ -ю есть дуга, 0 иначе
- Аналогично представляется размеченный граф, но  $a[i][j] \in R$

## □ Формат Matrix Market.

```
%>%MatrixMarket matrix coordinate real general
%=====
% Этот файл содержит разреженную MxN матрицу с L
% ненулевыми элементами в формате Matrix Market:
%
% +-----+
% |%%MatrixMarket | <-- заголовок
% |% | <-+
% | % comments | -- 0 или больше комментариев
% |% | <-+
% | M N L | <-- строк, столбцов, элементов
% | I1 J1 A(I1, J1) | <-+
% | I2 J2 A(I2, J2) |
% | I3 J3 A(I3, J3) |
% | . . . |
% | IL JL A(IL, JL) | -- L строк
% +-----+
%
% Индексы начинаются с 1, т.е. A(1,1) это первый элемент
%>%=====
```



## □ Сложные объявления. Правило улитки.

### Сложные объявления. Правила.

1. Читать по часовой стрелке, при этом отправной точкой является идентификатор.
2. Когда встречается очередной элемент объявления - заменяем его на слово из Таблицы 1.
3. Скобки "(" ")" могут использовать для изменения приоритета. Пока внутри скобок "(" ")" не прочитаны все элементы, "покидать" их нельзя.



## □ gets(), strcpy() и buffer overflow attack (sem02p10)

- Вообще, gets() была изъята из C11 и дальнейших версий, так как не может защитить от переполнения буфера, на замену ей была встроена fgets()

```
1 #include <stdio.h>
2 #include <string.h>
3
4 int buffer_overflow()
5 {
6 char pass[5];
7 int secret = 0xDEADBEEF;
8 gets(pass); //44444444444444444444
9 printf("%x", secret); //??
10 }
```

- Касательно strcpy() из string.h похожая история

```

#include <stdio.h>
#include <string.h>

int main(void) {
 setbuf(stdout, NULL);
 int secret = 0xDEADBEEF;
 char dst[5] = "dead";
 char src[100] = "beefbeefBBBBBBBBBBBBBBBBBBBBBB";
 strcpy(dst, src);
 printf("%s\n", dst);
 printf("%s\n", secret); // Затирается secret
 printf("%s\n", src); // eefBBBBBBBBBBBBBBBBBBBBBB
}

```

## ❑ Размещение строк и строковых констант в памяти (стек, куча) (sem02p12)

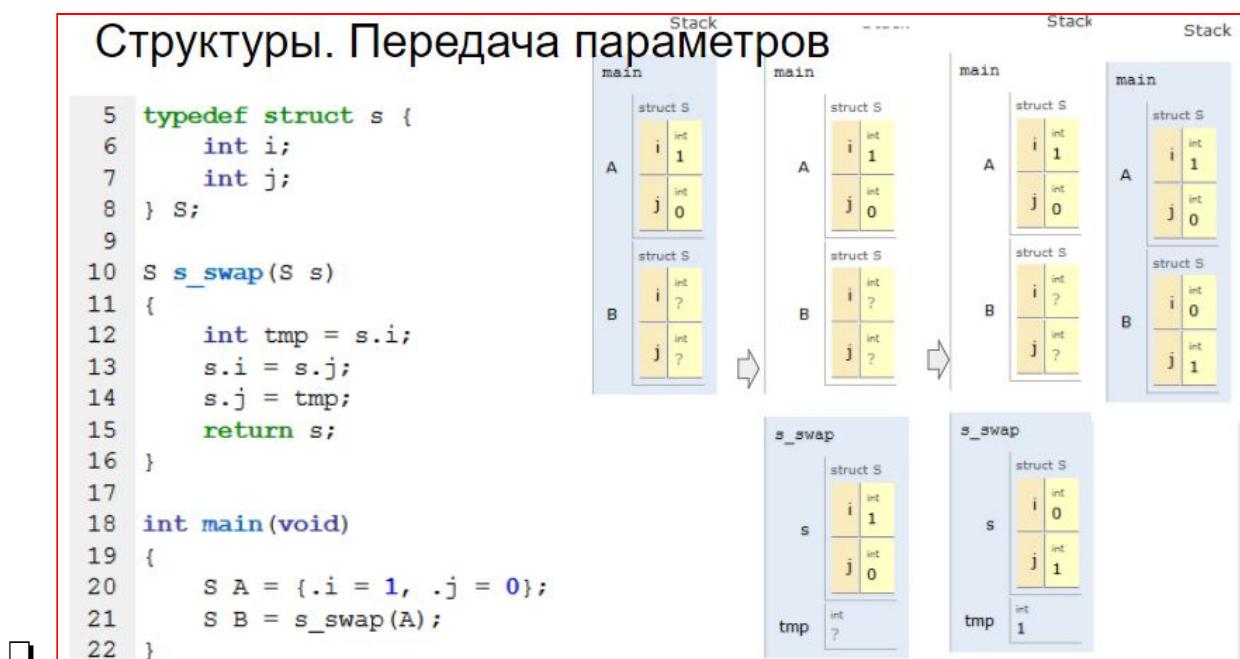
- ❑ Как обычный массив типа char, но в конце еще символ конца строки '\0'
- ❑ STACK

- ❑ char s[] = "String";
- ❑ char s[6] = "String";

## ❑ HEAP

- ❑ char\* s = (char\*) malloc(6 + 1);
- ❑ strcpy(s, "String");
- ❑ S[6] = '\0';

## ❑ Передача параметров типа struct (sem02p20)



## ❑ Списки смежности, представление графа. (sem03p14)

- ❑ P.S. подглядите из учебника по дискретной математике

## ❑ Флаги управления inline функциями (sem06p11)

- ❑ Можно получить выигрыш во времени, если дать компилятору копировать код функции непосредственно в код программы по месту вызова, а не создавать функцию в памяти. Такие функции называются встраиваемые (*inline*).  
Следует отметить, что *inline* - лишь рекомендация, а не команда компилятору заменять вызов функции ее телом. Он может подсчитать встраивание нецелесообразным и просто проигнорировать модификатор *inline* и трактовать функцию как обычную. Так что как говорится, на все воля компилятора... (прим.: на Windows существует модификатор `__forceinline`, который заставляет компилятор встроить функцию)
- ❑ Скорость:
  - ❑ быстрее - внедрение кода функции в код программы поможет избежать использования лишних инструкций (связанных с вызовом функции и возвратом из нее)
  - ❑ медленнее - слишком частое использование встраиваемых функций, к тому же больших, приведет к разрастанию кода (помимо этого, компилятор иногда вынужден использовать дополнительные временные переменные, чтобы сохранить семантику), что может привести в пробуксовке, т.е. в процессе работы программы операционной системе постоянно потребуется производить подкачку новых страниц
- ❑ Размер исполняемого файла:
  - ❑ увеличить - как уже упоминалось, слишком частое использование встраиваемых функций, к тому же больших, приведет к разрастанию кода и соответственно увеличению размера исполняемого файла
  - ❑ уменьшить - благодаря оптимизирующему компилятору размер исполняемого файла может и уменьшиться при использовании очень маленьких встраиваемых функций, так как компилятору при этом не нужно будет создавать "лишние" инструкции для вызова функции и выхода из нее, помещению аргументов в стек и обратно. Так же во время внедрения большой встраиваемой функции в код программы оптимизирующий компилятор может удалить избыточный код, что опять же может уменьшить размер файла.
- ❑ `__attribute__((always_inline))`
  - ❑ Если, анализируя свое приложение, вы видите в горячем месте какие-то незаинлайненные функции небольшого размера, то возможны две причины такой ситуации: компилятор не счел конкретный вызов выгодным для подстановки или решение о подстановке не было принято из-за превышения порога расширения кода. Повлиять на это можно, изменяя пороговые

значения или навешивания на функцию атрибуты типа `inline`. Поскольку изменения пороговых величин невозможно сделать для конкретной функции, то такое изменение вызовет увеличение размера всего приложения. То же самое можно сказать о атрибуте `inline`. Атрибут будет воздействовать как на «горячие», так и на «холодные» вызовы. При этом `inline` не гарантирует подстановку, если уже превышены пороги расширения кода. Более мощным атрибутом является `always_inline`. В этом случае функция всегда будет заинлайнена.

- ❑ `__attribute__((noinline))`
  - ❑ Не инлайнить данную функцию
- ❑ `-Winline`
  - ❑ Предупреждает, если функция не может быть сделана `inline` и при этом была объявлена `inline`, или же была дана опция '`-finline-functions`'.
- ❑ `-finline-functions-called-once (01)`
  - ❑ Эта опция позволяет компилятору рассмотреть возможность включения в вызывающий объект все статических функций, которые вызываются один раз, даже если они не были обозначены атрибутом как встроенные.  
This option is enabled at levels -O1, -O2, -O3, and -Os.
- ❑ `-finline-small-functions (02)`
  - ❑ Опция `-O2` включает в себя `-finline-small-functions`, и в этом случае `gcc` подставляет «маленькие» функции, подстановка которых, по идее, не сильно влияет на размер вызывающей функции.
- ❑ `-finline-functions (03)`
  - ❑ Интегрирует все простые функции в вызывающие функции. Компилятор эвристически решает, какие функции достаточно просты, чтобы их стоило интегрировать таким образом.
  - ❑ Если все вызовы к данной функции интегрированы, а функция объявлена `static`, тогда ассемблерный код функции обычно не выводится в своем настоящем виде.

## ❑ Пример unspecified behaviour в разных компиляторах (sem06p21)

- ❑ GCC 1.17 — запускала `nethack`, когда встречала в коде программы неизвестные `#pragma`
- ❑ GCC 4.6.3 - выполнение кода внутри `if (p) {...}` и `if (!p) {...}` когда `p` не инициализирована
- ❑ `i++ + ++i` - неопределенное поведение
- ❑ На некоторых компиляторах результат `MAX_INT + 1 > i` может давать в каждом случае разные значения

## ❑ Утилиты и их ключи использующиеся при статической/динамической компоновке

- ❑ Выше есть, в разделе библиотек
- ❑ \*-devel пакеты. \*.a, \*.o, \*.dll, \*.so файлы. (sem07)

~~Что такое пакет нужно объяснять?~~

~~Ну вкратце~~

Библиотеки которые можно скачать

И не только библиотеки

Обычно они качаются динамические

Тип что-бы кто угодно мог юзать

Но если ты захочешь запихнуть статически в своё  
приложение чего либо, дабы не париться с зависимостями

То тебе понадобится статическая либа

Они качаются отдельно в -devel пакетах

devel = development

Для разработчика

Конец



- ❑ Библиотека включает в себя

- ❑ заголовочный файл;
  - ❑ откомпилированный файл самой библиотеки:
  - ❑ библиотеки меняются редко – нет причин перекомпилировать каждый раз;
  - ❑ двоичный код предотвращает доступ к исходному коду.

- ❑ Библиотеки делятся на

- ❑ Статические
  - ❑ динамические
- ❑ \*.o - объектные файлы
- ❑ \*.dll - библиотеки на windows
- ❑ \*.so - библиотеки на linux

- ❑ Shared objects, soname, name (sem07p13)

- ❑ Разделяемые библиотеки (**shared libraries**) - это такие библиотеки, которые загружаются приложениями при запуске. Если такая библиотека установлена правильно, то все приложения, запускаемые после установки библиотеки, начинают использовать эту новую разделяемую библиотеку. На самом деле, всё гораздо гибче и совершеннее, ибо линуксовая их реализация позволяет:

- ❑ совершенствовать библиотеки, продолжая поддерживать программы, использующие более старые версии этих библиотек, не совместимые с новыми.
- ❑ заменять какие-то конкретные библиотеки или даже конкретные функции в библиотеке во время выполнения какой-то конкретной программы.
- ❑ проделывать всё это "на горячую", когда программа запущена и какие-то библиотеки уже используется.

- ❑ Имена разделяемых библиотек.

- ❑ Каждая разделяемая библиотека имеет специальное имя, называемое "soname". Оно имеет префикс "lib", само имя библиотеки, также слово ".so", за которым следует номер стадии и номер версии библиотеки, которые меняются каждый раз, когда у библиотеки меняется интерфейс. (Особое исключение: библиотеки низкого уровня языка С не начинаются с "lib"). Полное имя soname включает в качестве префикса имя директории, в которой библиотека находится. На живых системах полное имя библиотеки - это просто символическая ссылка на имя real name разделяемой библиотеки.

```
1 all: my_main.c
2
3
4 main:
5 gcc -o main main.c -L. -lmy
6 file-sample my_main.c
7 libmy:
8 gcc -fPIC -c libmy.c
9 ld -shared -soname libmy.so.1 -o libmy.so.1.0 -lc libmy.o
10
```

- ❑ Также, каждая разделяемая библиотека имеет имя real name. Это - имя файла, содержащего собственно код библиотеки. Real name в дополнение к soname содержит номер стадии, минорный номер (minor number), второй номер стадии и номер релиза. Второй номер и номер релиза не обязательны (are optional). Минорный номер и номер релиза дают вам знать, какие именно версии библиотек установлены у вас в системе.
- ❑ Кстати, в документации на библиотеку эти два последних номера могут с библиотечными своими собратьями не совпадать.
- ❑ Далее. У библиотеки существует ещё один тип имени, который используется системой, когда библиотеку запрашивают. Называется оно linker name, которое на самом деле равно soname, но без указания версии.

- ❑ Способ содержания разделяемых библиотек в хозяйстве состоит в различении их по именам. Каждое приложение должно помнить о разделяемой библиотеке, которая ему нужна, помня только soname этой библиотеки. Наоборот, когда вы разделяемую библиотеку создаете, вы имеете дело только с именем файла, содержащего более детальную информацию о версии. Когда вашу библиотеку будут устанавливать, её запишут в один из специально предназначенных для этого каталогов в файловой системе и запустят программу ldconfig. Эта хитрая программа смотрит в упомянутые специальные каталоги и создаёт soname-имена как символические ссылки к реальным именам, публикуя новости с фронтов в файле /etc/ld.so.cache (речь о нём позже).
- ❑ ldconfig не создаёт linker-имена. Обычно это делается при инициализации библиотеки и linker-имена создаются просто как символические ссылки к последним актуальным soname-именам или к последним актуальным real-именам. Я бы рекомендовал делать объектом ссылки soname-имена, ибо меняются они реже, чем real-имена. А причина, по которой ldconfig автоматически не пропускает linker-имена в том, чтобы дать юзерам возможность запустить код, который выбирает linker-имя библиотеки на свое усмотрение.
- ❑ Таким образом, /usr/lib/libreadline.so.3 - это полное имя soname, которое на самом деле является символической ссылкой на реальное имя типа /usr/lib/libreadline.so.3.0. Linker-имя будет выглядеть как /usr/lib/libreadline.so, являющееся символической ссылкой на /usr/lib/libreadline.so.3

### Shared objects. soname и name

- Linker name = <имя\_библиотеки>
- Soname это = lib<имя\_библиотеки>.so.<версия>
- Fully qualified soname = <полный\_путь\_до\_файла> + soname
- Real name = <soname>.<минорная\_версия>.<релизная\_версия>

soname и linker name - ссылка на real name

Программы внутри себя используют soname (поле DT\_NEEDED)

soname хранится внутри .so файла в поле DT SONAME

Linker name создается установщиком (пакетным менеджером)

## **Установка и использование разделяемой библиотеки.**

Проще всего - скопировать в один из стандартных каталогов (например /usr/lib) и запустить ldconfig(8).

Сначала вы должны создать разделяемую библиотеку где-либо. Затем вам понадобится создать необходимые символические ссылки, в частности ссылку soname, показывающую на реальное имя (также ссылку soname, на которой нет версии, то есть, soname оканчивающееся на ".so" для пользователей, которым версия вокруг табуретки). Простейший способ:

```
ldconfig -n каталог с разделяемыми библиотеками
```

Наконец, когда вы собираете ваши программы, вам нужно указать линкеру используемые вами статические и разделяемые библиотеки. Для этого есть флаги -l и -L. Если вы не устанавливаете библиотеку в какой-то из стандартных каталогов, вам нужно оставить их где угодно, но показать на них своей программе. Это делается несколькими способами: флаг gcc "-L", флаг "-rpath", переменные окружения (например LD\_LIBRARY\_PATH, содержащий список каталогов, разделённых двоеточием, в которых будут предприниматься попытки найти нужную разделяемую библиотеку перед попытками искать её в стандартных каталогах). Если вы используете bash, то можете вызывать свою программу так:

```
LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH my_program
```

Если своей библиотекой вы хотите заменить только часть функций, можете сделать это так: создать замещающий объектный файл и установить LD\_PRELOAD. Функции из этого объектного файла заменят только те же самые функции (не касаясь остальных).

Обычно вы можете заменять библиотеки без специальных предупреждений. Если новая библиотека реализует изменения в API, тогда автор библиотеки меняет soname. Таким образом, в одной системе могут существовать несколько библиотек, но каждая программа в системе находит нужные ей. Если же программа сломалась при замене библиотеки, при которой сохранилось старое её soname, вы можете заставить программу использовать старую версию библиотеки. Для этого скопируйте старую библиотеку куда-нибудь, затем переименуйте программу (например прибавьте к старому имени ".orig"), после чего создайте небольшой "обёрточный" скрипт. Он будет устанавливать переменные окружения, сообщающие загрузчику альтернативный путь для поиска библиотек и затем запускать переименованную копию программы. Вот такой:

```
#!/bin/sh
export LD_LIBRARY_PATH=/usr/local/my_lib:$LD_LIBRARY_PATH
exec /usr/bin/my_program.orig &
```

Пожалуйста, не вооружайтесь этим способом запуска программ, когда пишете свои собственные программы. Попытайтесь гарантировать, что ваши новые библиотеки совместимы со старыми версиями или не забывайте увеличивать номер версии в soname-имени при каждом изменении, нарушающем совместимость. Описанный метод рассматривайте как помощь в исключительных ситуациях.

Просмотреть список разделяемых библиотек, используемых программой возможно с помощью ldd(1). Например так:

```
ldd /bin/ls
```

В основном вы увидите список soname-имён в парах с полными именами файлов. Практически все ваши программы будут зависеть от:

- \* `/lib/ld-linux.so.N` (где N больше единицы, но обычно 2). Эта библиотека загружает все другие библиотеки.
- \* `libc.so.N` (где N больше или равно шести). Это стандартная библиотека языка С. Даже другие языки программирования часто используют `libc` (как минимум для реализации своих библиотек).

Но не запускайте `ldd` для тех программ, которым не доверяете. Как ясно изложено в руководстве по `ldd(1)`, она выполняет свои функции, устанавливая специальные переменные окружения (для ELF объектов `LD_TRACE_LOADED_OBJECTS`), после чего запускает программу. Таким образом ненадёжной программе может быть предоставлена возможность запуска произвольного кода, от имени пользователя, выполняющего `ldd`.



## □ DLL/dependency hell (sem07p17)

- DLL hell (DLL-кошмар, буквально: DLL-ад) — тупиковая ситуация, связанная с управлением динамическими библиотеками DLL в операционной системе Microsoft Windows. Аналогичная проблема в других ОС носит название Dependency hell.  
Сущность проблемы заключается в конфликте версий DLL, призванных поддерживать определённые функции. DLL hell — пример плохой концепции программирования, которая, подобно скрытой мине, приводит к резкому возрастанию трудностей при усложнении и совершенствовании системы.
- По исходному замыслу, DLL должны быть совместимыми от версии к версии и взаимозаменяемыми в обе стороны.
- Реализация механизма DLL такова, что несовместимость и невзаимозаменяемость становится скорее правилом, чем исключением, что приводит к большому количеству проблем.
  - **Отсутствие** стандартов на имена, версии и положение DLL в файловой структуре приводит к тому, что несовместимые DLL легко замещают или отключают друг друга.
  - **Отсутствие** стандарта на процедуру установки приводит к тому, что установка новых программ приводит к замещению работающих DLL на несовместимые версии.
  - **Отсутствие** поддержки DLL со стороны компоновщиков и механизмов защиты приводит к тому, что несовместимые DLL могут иметь одинаковые имя и версию.
  - **Отсутствуют** стандартные инструменты идентификации и управления системой DLL пользователями и администраторами.
  - **Использование** отдельных DLL для обеспечения связи между задачами приводит к нестабильности сложных приложений.

❑ Для избежания конфликтов обычно используют множество избыточных копий DLL для каждого приложения, что сводит на нет исходную идею получения преимущества от DLL как стандартных модулей, хранящихся один раз в памяти и разделяемых многими задачами. Кроме того, при таком опыте после исправления ошибок в DLL или восстановления системы из архива количество различных DLL, носящих одно и то же имя и выполняющих те же функции, возрастает, а автоматическое обновление версии или исправление ошибок становится невозможным.

- ❑ main.exe зависит от version-0.3.dll и bar.dll. bar в свою очередь, зависит от version-0.2.dll, которая бинарно не совместима с версией 0.3 (не просто символы отсутствуют, а совпадают имена, но различное число аргументов, или создают объекты разной природы и т. п.). Затрут ли символы из version-0.2.dll оные из version-0.3.dll? Тот же вопрос стоит тогда, когда используется одна статическая версия библиотеки (скажем, version-0.2.lib) и динамическая (version-0.3.dll);
- ❑ создание перемещаемых приложений: где динамический загрузчик будет искать version-0.? dll и bar.dll для приложения из предыдущего пункта? Найдёт ли он зависимости main.exe, если тот будет перемещён в другую папку? Как нужно собрать main.exe, чтобы зависимости искались относительно исполняемого файла?
- ❑ dependency hell: две версии одной библиотеки /opt/kde3/lib/libkdecore.so и /opt/kde4/lib/libkdecore.so (с которой плазма уже не падает), половина программ требуют первую, другая половина программ — вторую. Обе библиотеки нельзя поместить в одну область видимости (один каталог). Эта же проблема есть и в п. 1, т. к. надо поместить две версии библиотеки version в один каталог.

## ❑ Побег из ада

- ❑ Ответ прост: надо добавить версию в имя файла библиотеки. Это позволит размещать файлы библиотек в одном каталоге. При этом рекомендуется добавлять версию ABI, а не API, порождая тем самым две параллельных ветки версий и соответствующие трудности.
- ❑ Контроль версии – очень рутинная работа. Рассмотрим схему x.y.z:
  - ❑ x – мажорный выпуск. Ни о какой совместимости речи не идёт;
  - ❑ у – минорный выпуск. Либо совместима на уровне исходных текстов, но двоичная совместимость может быть сломана;
  - ❑ z – багофикс. Либо совместима в обе стороны.

- ❑ Тогда в имя файла разумно включить х.у. Если при увеличении минорной версии совместимость сохранили, то достаточно сделать соответствующий симлинк:
    - ❑ version-1.1.dll
    - ❑ version-1.0.dll → version-1.1.dll
  - ❑ Будут работать и приложения, использующие version-1.1.0, и те, кто использует version-1.0.x.
  - ❑ Если совместимость сломали, то в системе будет два файла и снова всё будет работать.
  - ❑ Если по каким-то причинам совместимость сломали при багофиксе, то должна быть увеличена минорная версия (и нечего фейлится, как это сделала команда любимейшего Qt).
  - ❑ Кстати говоря, никто не запрещает вообще включить версию API – тогда символьических ссылок будет больше, т.к. совместимость чаще сохраняется. Зато в этом случае упомянутый файл Qt разрулился бы легко и не заставил увеличивать минорную версию.
  - ❑ Это справедливо для всех платформ.
  - ❑ Решение оставшихся двух вопросов отличается в зависимости от ОС.
- ❑ **Format string attack, buffer overflow attack (lab00)**
- ❑ Ниже пример

---

## Insights

```
#include <stdio.h>

int test(int a, int *p) {
 int d = 0xBADDCAFE;
 printf("test:a: 0x%08x %p \n", *((&a) + 1), &d);
 // print CAFEEFACE using p variable
}

int main() {
 setbuf(stdout, NULL);

 int a = 0xDEADBEEF;
 int b = 0x77777777;
 int c = 0xCAFEFACE;
 int * p = &b; // теперь тут лежит 0xFFFFFFFF (адрес, по которому лежит b)
```

```
printf("0x%08x \n", *((&b) + 1)); // 0xcafeface
// увеличиваем указатель на 1 и попадаем в следующую переменную
printf("0x%08x \n", *((&b) - 1)); // 0x9214acf0

printf("a=%08x b=%08x c=%08x \n", a, b, c); // a=deadbeef b=77777777 c=cafeface
printf("%d (or 0x%08x) at %p \n", *p, *p, p); // 2004318071 (or 0x77777777) at
0x7ffec242e22c

// Объявлены 3 переменный a, b, c
// переменной a выделяется память
// 0xFFFFFFFF DE AD BE EF int a
// 0xFFFFFFFFD 77 77 77 77 int b

test(a, &a); // test:a: 0x00000000 0x7ffea53adbec
// различаются на 16

printf("%s", "Hello world \n");
// спецификатор s ждет для обработки указатель на массив символов
// где то в памяти выделяется место под строки и на стек
// кладутся 2 указателя

printf("%s%s \n"); // Segmentation fault (core dumped);
// обращение по нулевому адресу

printf("016llx\n"); // !x64 - %016llx, x86 - %08x
printf("%016llx %016llx %016llx %016llx %016llx %016llx\n");
// 016llx
// 00007f26ad352663 00007f26ad353740 ffffffffffffffff 0000000000000006 0000000000000000
00000000004006e0 deadbeef77777777
// чувак хочет вывести лонг лонг, а так как мы ничего не передали
// он прочитает из памяти то, что лежит в памяти, а выше по стеку лежат
// переменные. каждый спецификатор печатает из стека 48 байт из стека
// таким образом можно делать эксплойты и вытаскивать информацию из памяти

int a = 0xDEADBEEF;
int b = 0x77777777;
int c = 0xBADDFOOD;

char* secret = "pa$$w0rd"; // на стеке выделяется место только под указатель
char* user = malloc(100); // выделяем сто байт, в user кладется указатель на начало
// выделенных сто байт

fgets(user); // get string читаем все из консоли
// данные помещаются в буфер из 100 байт, которые мы выделили
printf(user);
// указатель занимает 4 байта
// можно отправить через консоль %llx %llx %llx %llx и увидеть памяти
return 0;
}
```

