

Программа на языке Си состоит из набора директив препроцессора, определений функций и глобальных объектов. Директивы препроцессора управляют преобразованием текста до его компиляции. Глобальные объекты определяют используемые данные или состояние программы. А функции определяют поведение или действия программы. Точкой входа для выполнения программы является функция с фиксированным идентификатором `main`. Язык Си является блочно-структурированным. Каждый блок заключается в фигурные скобки. Каждое действие в языке Си заканчивается символом «точка с запятой» — `;`. В качестве действия может выступать вызов функции или осуществление некоторых операций.

Константа — это величина, которая при выполнении программы остаётся неизменной. Переменная — это ячейка памяти для временного хранения данных. `int a` - переменная, `'с'` - символьная константа. Константу можно задать с помощью квалификатора `const`, макроса `#define` и перечисления `enum`.

Компиляция – это процесс превращения исходного кода в машинный код. Этапы компиляции:

1) Препроцессинг

Препроцессор — это макро процессор, который преобразовывает вашу программу для дальнейшего компилирования. На данной стадии происходит работа с препроцессорными директивами. Например, препроцессор добавляет хэдеры в код (`#include`), убирает комментирования, заменяет макросы (`#define`) их значениями, выбирает нужные куски кода в соответствии с условиями `#if`, `#ifdef` и `#ifndef`.

2) Компиляция

На данном шаге `gcc` выполняет свою главную задачу — компилирует, то есть преобразует полученный на прошлом шаге код без директив в ассемблерный код. Это промежуточный шаг между высокоуровневым языком и машинным (бинарным) кодом.

Ассемблерный код — это доступное для понимания человеком представление машинного кода.

3) Ассемблирование

Ассемблер преобразовывает ассемблерный код в машинный код, сохраняя его в объектном файле.

Объектный файл — это созданный ассемблером промежуточный файл, хранящий кусок машинного кода.

4) Компоновка

Компоновщик (линкер) связывает все объектные файлы и статические библиотеки в единый исполняемый файл, который мы и сможем запустить в дальнейшем. Для того, чтобы понять как происходит связка, следует рассказать о таблице символов.

Таблица символов — это структура данных, создаваемая самим компилятором и хранящаяся в самих объектных файлах. Таблица символов хранит имена переменных, функций, классов, объектов и т.д., где каждому идентификатору (символу) соотносится его тип, область видимости. Также таблица символов хранит адреса ссылок на данные и процедуры в других объектных файлах.

2. Типы данных в языке Си. Арифметические операторы.

Машинное представление чисел. Примеры.

Язык Си предоставляет множество базовых типов. Большинство из них формируется с помощью одного из четырёх арифметических спецификаторов типа, (char, int, float и double), и опциональных спецификаторов (signed, unsigned, short и long).

В большинстве систем char - 1 байт, int - 4 байта, float - 8 байт, double - 16 байт.

char - символ, int - целое число, float - число с плавающей точкой, double - float с удвоенной точностью.

signed/unsigned - знаковый/беззнаковый, short/long -

короткий/длинный (в 2 раза меньше/больше байт занимает в памяти).

void - тип без значения, _Bool - логический тип.

Структура (struct) - сложный тип данных, состоящий из нескольких полей возможно разных типов.

Арифметические операторы: + — сложение; - — вычитание; * — умножение; / — деление; % — остаток от деления.

Постфиксная/префиксная инкрементация/декрементация: x++, ++x, x--, --x.

```
int x = 0;
```

```
x++ = 0, x = 1
```

```
++x = 1, x = 1
```

Целые числа:

Прямой код представляет собой одинаковое представление значимой части числа для положительных и отрицательных чисел и отличается только знаковым битом. В прямом коде число 0 имеет два представления «+0» и «-0».

Обратный код для положительных чисел имеет тот же вид, что и прямой код, а для отрицательных чисел образуется из прямого кода положительного числа путем инвертирования всех значащих разрядов прямого кода. В обратном коде число 0 также имеет два

представления «+0» и «-0».

Дополнительный код для положительных чисел имеет тот же вид, что и прямой код, а для отрицательных чисел образуется путем прибавления 1 к обратному коду. Добавление 1 к обратному коду числа 0 дает единое представление числа 0 в дополнительном коде. Однако это приводит к асимметрии диапазонов представления чисел относительно нуля.

Числа с плавающей запятой:

Знак		Порядок						Мантисса										
0	0																	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
14		10						9	0									

Знак — один бит, указывающий знак всего числа с плавающей точкой. Порядок и мантисса — целые числа, которые вместе со знаком дают представление числа с плавающей запятой в следующем виде:

$(-1)^S \times M \times B^E$, где S — знак, B — основание, E — порядок, а M — мантисса. Десятичное число, записываемое как $R \times E$, где R — число в полуинтервале $[1;10)$, E — степень, в которой стоит множитель 10 ; в нормализованной форме модуль R будет являться мантиссой, а E — порядком, а S будет равно 1 тогда и только тогда, когда R принимает отрицательное значение.

Например, в числе $-2435e9$

$S = 1$

$B = 10$

$M = 2435$

$E = 9$

3. Логические операторы и операторы отношения. Ветвления.

Тернарный оператор. Примеры.

Результатом логической операции является либо 0, либо 1. Результат имеет тип `int`. `&&` - логическое и, `||` - логическое или.

Результатом реляционного выражения является 1, если проверенная связь имеет значение `true`, и 0, если значение `false`. Результат имеет тип `int`.

`<` Первый операнд меньше второго операнда

`>` Первый операнд больше второго операнда

`<=` Первый операнд меньше или равен второму операнду



`>=` Первый операнд больше или равен второму операнду

== Первый операнд равен второму операнду

!= Первый операнд не равен второму операнду

Условный оператор if

Условный оператор **if** может использоваться в форме **полной** или **неполной** развилки.

Неполная развилка	Полная развилка
<pre>1 if (Условие) 2 { 3 БлокОпераций1; 4 }</pre>	<pre>1 if (Условие) 2 { 3 БлокОпераций1; 4 } 5 else 6 { 7 БлокОпераций2; 8 }</pre>
	

В случае неполной развилки если **Условие** истинно, то **БлокОпераций1** выполняется, если **Условие** ложно, то **БлокОпераций1** не выполняется.

Оператор ветвления switch (оператор множественного выбора)

Оператор `if` позволяет осуществить выбор только между двумя вариантами. Для того, чтобы производить выбор одного из нескольких вариантов необходимо использовать вложенный оператор `if`. С этой же целью можно использовать оператор ветвления `switch`.

Общая форма записи

```
switch (ЦелоеВыражение)
{
    case Константа1: БлокОпераций1;
        break;
    case Константа2: БлокОпераций2;
        break;
    . . .
    case Константаn: БлокОперацийn;
        break;
    default: БлокОперацийПоУмолчанию;
        break;
}
```

Оператор ветвления `switch` выполняется следующим образом:

- ❑ вычисляется `ЦелоеВыражение` в скобках оператора `switch`;
- ❑ полученное значение сравнивается с метками (Константами) в опциях `case`, сравнение производится до тех пор, пока не будет найдена метка, соответствующая вычисленному значению целочисленного выражения;
- ❑ выполняется `БлокОпераций` соответствующей метки `case`;
- ❑ если соответствующая метка не найдена, то выполнится `БлокОперацийПоУмолчанию`, описанный в опции `default`.

Тернарные операции

Тернарная условная операция имеет 3 аргумента и возвращает свой второй или третий операнд в зависимости от значения логического выражения, заданного первым операндом. Синтаксис тернарной операции в языке Си

```
Условие ? Выражение1 : Выражение2;
```

Если выполняется `Условие`, то тернарная операция возвращает `Выражение1`, в противном случае - `Выражение2`.

4. Циклы. Средства управления циклами. Примеры.

Циклом называется блок кода, который для решения задачи требуется повторить несколько раз.

Каждый цикл состоит из блока проверки условия повторения цикла и тела цикла.

Цикл с предусловием while

Общая форма записи

```
while (Условие)
{
    БлокОпераций;
}
```

Если `Условие` выполняется (выражение, проверяющее `Условие`, не равно нулю), то выполняется `БлокОпераций`, заключенный в фигурные скобки, затем `Условие` проверяется снова.

Цикл с постусловием `do...while`

Общая форма записи

```
do {  
    БлокОпераций;  
} while (Условие);
```

Цикл `do...while` — это цикл с постусловием, где истинность выражения, проверяющего `Условие` проверяется после выполнения `Блока Операций`, заключенного в фигурные скобки. Тело цикла выполняется до тех пор, пока выражение, проверяющее `Условие`, не станет ложным, то есть тело цикла с постусловием выполнится хотя бы один раз.

Использовать цикл `do...while` лучше в тех случаях, когда должна быть выполнена хотя бы одна итерация, либо когда инициализация объектов, участвующих в проверке условия, происходит внутри тела цикла.

Параметрический цикл `for`

Общая форма записи

```
for (Инициализация; Условие; Модификация)  
{  
    БлокОпераций;  
}
```

`for` — параметрический цикл (цикл с фиксированным числом повторений). Для организации такого цикла необходимо осуществить три операции:

- ❑ **Инициализация** - присваивание параметру цикла начального значения;
- ❑ **Условие** - проверка условия повторения цикла, чаще всего - сравнение величины параметра с некоторым граничным значением;
- ❑ **Модификация** - изменение значения параметра для следующего прохождения тела цикла.

Эти три операции записываются в скобках и разделяются точкой с запятой `;;`. Как правило, параметром цикла является целочисленная переменная.

Инициализация параметра осуществляется только один раз — когда цикл `for` начинает выполняться.

Проверка `Условия` повторения цикла осуществляется перед каждым возможным выполнением тела цикла. Когда выражение, проверяющее `Условие` становится ложным (равным нулю), цикл завершается.

Модификация параметра осуществляется в конце каждого выполнения тела цикла. Параметр может как увеличиваться, так и уменьшаться.

Операторы прерывания и продолжения цикла `break` и `continue`

В теле любого цикла можно использовать операторы прерывания цикла - `break` и продолжения цикла - `continue`.

Оператор `break` позволяет выйти из цикла, не завершая его.

Оператор `continue` позволяет пропустить часть операторов тела цикла и начать новую итерацию.

5. Указатели в языке Си. Арифметика указателей. Примеры.

Указатель — переменная, содержащая адрес объекта.

Общая форма объявления указателя

```
тип *ИмяОбъекта;
```

Тип указателя— это тип переменной, адрес которой он содержит. Для работы с указателями в Си определены две операции: операция * (звездочка) — позволяет получить значение объекта по его адресу — определяет значение переменной, которое содержится по адресу, содержащемуся в указателе; операция & (амперсанд) — позволяет определить адрес переменной. Арифметика указателей: увеличение на единицу означает, что мы хотим перейти к следующему объекту в памяти, который находится за текущим и на который указывает указатель. А уменьшение на единицу означает переход назад к предыдущему объекту в памяти. Аналогично указатель будет изменяться при прибавлении/вычитании не единицы, а какого-то другого числа. В отличие от сложения операция вычитание может применяться не только к указателю и целому числу, но и к двум указателям одного типа.

6. Массивы в языке Си. Примеры.

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

Массив характеризуется следующими основными понятиями:

Элемент массива (значение элемента массива) – значение, хранящееся в определенной ячейке памяти, расположенной в пределах массива, а также адрес этой ячейки памяти.

Каждый элемент массива характеризуется тремя величинами: адресом элемента — адресом начальной ячейки памяти, в которой расположен этот элемент; индексом элемента (порядковым номером элемента в массиве); значением элемента.

Адрес массива – адрес начального элемента массива.

Объявление и инициализация массивов

Для объявления массива в языке Си используется следующий синтаксис:

тип имя[размерность]={инициализация};

Инициализация представляет собой набор начальных значений элементов массива, указанных в фигурных скобках, и разделенных запятыми.

```
int a[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}; // массив a из 10 целых чисел
```

Если количество инициализирующих значений, указанных в фигурных скобках, меньше, чем количество элементов массива, указанное в квадратных скобках, то все оставшиеся элементы в массиве (для которых не хватило инициализирующих значений) будут равны нулю. Это свойство удобно использовать для задания нулевых значений всем элементам массива.

```
int b[10] = {0}; // массив b из 10 элементов, инициализированных 0
```

Если массив проинициализирован при объявлении, то константные начальные значения его элементов указываются через запятую в фигурных скобках. В этом случае количество элементов в квадратных скобках может быть опущено.

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

Многомерные массивы

В языке Си могут быть также объявлены многомерные массивы. Отличие многомерного массива от одномерного состоит в том, что в одномерном массиве положение элемента определяется одним индексом, а в многомерном — несколькими. Примером многомерного массива является матрица.

Общая форма объявления многомерного массива

```
тип имя[размерность1][размерность2]...[размерностьm];
```

Передача массива в функцию

Обработку массивов удобно организовывать с помощью специальных функций. Для обработки массива в качестве аргументов функции необходимо передать адрес массива и размер массива.

Исключение составляют функции обработки строк, в которые достаточно передать только адрес.

При передаче переменные в качестве аргументов функции данные передаются как копии. Это означает, что если внутри функции произойдёт изменение значения параметра, то это никак не повлияет на его значение внутри вызывающей функции.

7. Разработка пользовательских функций в языке Си. Примеры.

Все функции, в том числе и те, которые пишет пользователь, устроены сходным образом. У них имеется две основных составных части: заголовок функции и тело функции. Заголовок состоит из трёх обязательных частей:

тип возвращаемого значения;

имя функции;

параметры функции.

Параметры - это переменные, которые используются при создании функции.

Аргументы - это фактические значения (данные), которые передаются функции при вызове.

Функция, вызывающая саму себя называется рекурсивной.

Перед вызовом функции, необходимо объявить её прототип.

8. Структуры и перечисления в языке Си. Примеры.

Структура — это объединение нескольких объектов, возможно, различного типа под одним именем, которое является типом структуры. В качестве объектов могут выступать переменные, массивы, указатели и другие структуры.

Общая форма объявления структуры:

```
struct ИмяСтруктуры
{
    тип ИмяЭлемента1;
    тип ИмяЭлемента2;
    . . .
    тип ИмяЭлементап;
};
```

После закрывающей фигурной скобки } в объявлении структуры обязательно ставится точка с запятой.

Инициализация полей структуры

Инициализация полей структуры может осуществляться двумя способами:

- присвоение значений элементам структуры в процессе объявления переменной, относящейся к типу структуры;
- присвоение начальных значений элементам структуры с использованием функций ввода-вывода (например, printf() и scanf()).

В первом способе инициализация осуществляется по следующей форме:

```
struct ИмяСтруктуры ИмяПеременной={ЗначениеЭлемента1, ЗначениеЭлемента_2, . . . , ЗначениеЭлементап};
```

Имя структурной переменной может быть указано при объявлении структуры. В этом случае оно размещается после закрывающей фигурной скобки }. Область видимости такой структурной переменной будет определяться местом описания структуры.

```
struct complex_type // имя структуры
{
    double real;
    double imag;
} number; // имя структурной переменной
```

Обратиться к полю структуры можно с помощью оператора. Если нужно обратиться к полю указателя на структуру есть 2 варианта:

pb->title \equiv **(*pb).title**

Перечисления - это набор именованных целочисленных констант, определяющий все допустимые значения, которые может принимать переменная.

Перечисления определяются с помощью ключевого слова enum, которое указывает на начало перечисляемого типа. Стандартный вид перечислений следующий:

```
enum ярлык { список перечислений } список переменных;
```

По умолчанию, первое поле структуры принимает численное значение 0, следующее 1, следующее 2 и т.д. Можно задать нулевое значение явно. Перечисления используются для большей типобезопасности и ограничения возможных значений переменной.

9. Идентификаторы в языке Си. Области видимости
идентификаторов. Связывание. Примеры.

"Идентификаторы" или "символы" — это имена, задаваемые в программе для переменных, типов, функций и меток. Написание и

регистр символов в именах идентификаторов должны отличаться от всех ключевых слов. Вы не можете использовать ключевые слова (C или Microsoft) в качестве идентификаторов; они зарезервированы для специального использования.

Область видимости описывает участок или участки программы, где можно обращаться к идентификатору. Переменная в C имеет одну из следующих областей видимости:

в пределах блока, в пределах функции, в пределах прототипа функции и в пределах файла.

В пределах блока - идентификаторы, объявленные в {}.

В пределах функции - только метки go to.

В пределах прототипа функции - объявленные в прототипе функции.

В пределах файла - объявленные вне функций (глобальные переменные).

Переменная в C имеет одно из следующих связываний: внешнее связывание, внутреннее связывание или отсутствие связывания.

Переменные с областью видимости в пределах блока, функции или прототипа функции не имеют связывания. Это означает, что они являются закрытыми для блока, функции или прототипа, в котором определены. Переменная с областью видимости в пределах файла может иметь либо внутреннее, либо внешнее связывание. Переменная с внешним связыванием может применяться в любом месте многофайловой программы, а переменная с внутренним связыванием — где угодно в единице трансляции.

Как же тогда выяснить, внутреннее или внешнее связывание имеет переменная с областью видимости в пределах файла? Вы должны посмотреть, используется ли во внешнем определении спецификатор класса хранения `static`:

```
int giants = 5;           // область видимости в пределах файла,  
                           // внешнее связывание  
static int dodgers = 3;   // область видимости в пределах файла,  
                           // внутреннее связывание  
  
int main()  
{  
    ...  
}
```

Переменная `giants` может применяться в других файлах, которые представляют собой составные части той же самой программы. Переменная `dodgers` является закрытой для данного конкретного файла, но может использоваться любой функцией в этом файле.

10. Программные объекты. Продолжительность хранения объекта. Классы памяти. Примеры.

Продолжительность хранения характеризует постоянство объектов, доступных через идентификаторы. Объект в C имеет одну из

следующих четырёх продолжительностей хранения: статическую, потоковую, автоматическую или выделенную.

Если объект имеет статическую продолжительность хранения, он существует на протяжении времени выполнения программы.

Переменные с областью видимости в пределах файла имеют статическую продолжительность хранения.

Переменные с областью видимости в пределах блока обычно имеют автоматическую продолжительность хранения. Память для этих переменных выделяется, когда поток управления входит в блок, где они определены, и освобождается, когда поток управления покидает этот блок. Идея заключается в том, что память, используемая для автоматических переменных, является рабочим пространством или временной памятью, которая может применяться многократно.

Например, после завершения вызова функции память, которую функция использовала для своих переменных, может быть задействована при вызове следующей функции.

Массивы переменной длины демонстрируют небольшое исключение в том, что они существуют от места своего объявления и до конца блока, а не от начала блока и до его конца.

Переменная может иметь область видимости в пределах блока, но статическую продолжительность хранения. Чтобы создать такую переменную, объявите её внутри блока и добавьте в объявление ключевое слово `static`.

Таблица 12.1. Пять классов хранения

Класс хранения	Продолжительность хранения	Область видимости	Связывание	Объявление
Автоматический	Автоматическая	В пределах блока	Нет	В блоке
Регистровый	Автоматическая	В пределах блока	Нет	В блоке с указанием ключевого слова <code>register</code>
Статический с внешним связыванием	Статическая	В пределах файла	Внешнее	За рамками всех функций
Статический с внутренним связыванием	Статическая	В пределах файла	Внутреннее	За рамками всех функций с указанием ключевого слова <code>static</code>
Статический без связывания	Статическая	В пределах файла	Нет	В блоке с указанием ключевого слова <code>static</code>

Переносимые объектные файлы создаются на этапе компиляции в машинный код в ходе сборки проекта. Они считаются промежуточными и используются в качестве ингредиентов для создания дальнейших и конечных продуктов.

В переносимом объектном файле, полученном из скомпилированной единицы трансляции, можно найти следующие элементы:

- инструкции машинного уровня, сгенерированные из функций, найденных в единице трансляции (код);
- значения инициализированных глобальных переменных, объявленных в единице трансляции (данные);
- таблицу символов, содержащую все символы, которые были объявлены и на которые ссылается единица трансляции.

ELF формат - Executable and Linkable Format - состоит из секций.

Секция .text содержит все машинные инструкции для единицы трансляции. В секциях .data и .bss находятся соответственно значения для инициализированных глобальных переменных и количество байтов, необходимых для неинициализированных глобальных переменных. Секция .symtab хранит таблицу символов. символы в переносимых объектных файлах не обладают итоговыми, абсолютными адресами; данная информация определяется на этапе компоновки.

Исполняемые объектные файлы содержат больше секций, содержащих данные, необходимые для загрузки и выполнения программы.

Статическая библиотека в Unix — обычный архив с переносимыми объектными файлами.

В системах семейства Unix применительно к статическим библиотекам действует общепринятое соглашение об именовании.

Имя файла должно начинаться с lib и иметь расширение .a.

Команда для создания стат. библиотеки.

```
ar crs libexample.a aa.o bb.o ... zz.o
```

Терминал 3.14. Компоновка со статической библиотекой, созданной в примере 3.2

```
$ gcc main.o -L/opt/geometry -lgeometry -lm -o ex3_3.out
$
```

Параметр -L/opt/geometry сообщает компилятору gcc, что каталог /opt/geometry входит в число тех мест, в которых можно найти статические и разделяемые библиотеки. По умолчанию компоновщик ищет библиотечные файлы в традиционных каталогах, таких как /usr/lib или /usr/local/lib.

Параметр -lgeometry говорит компилятору gcc, что ему нужно искать файл libgeometry.a или libgeometry.so.

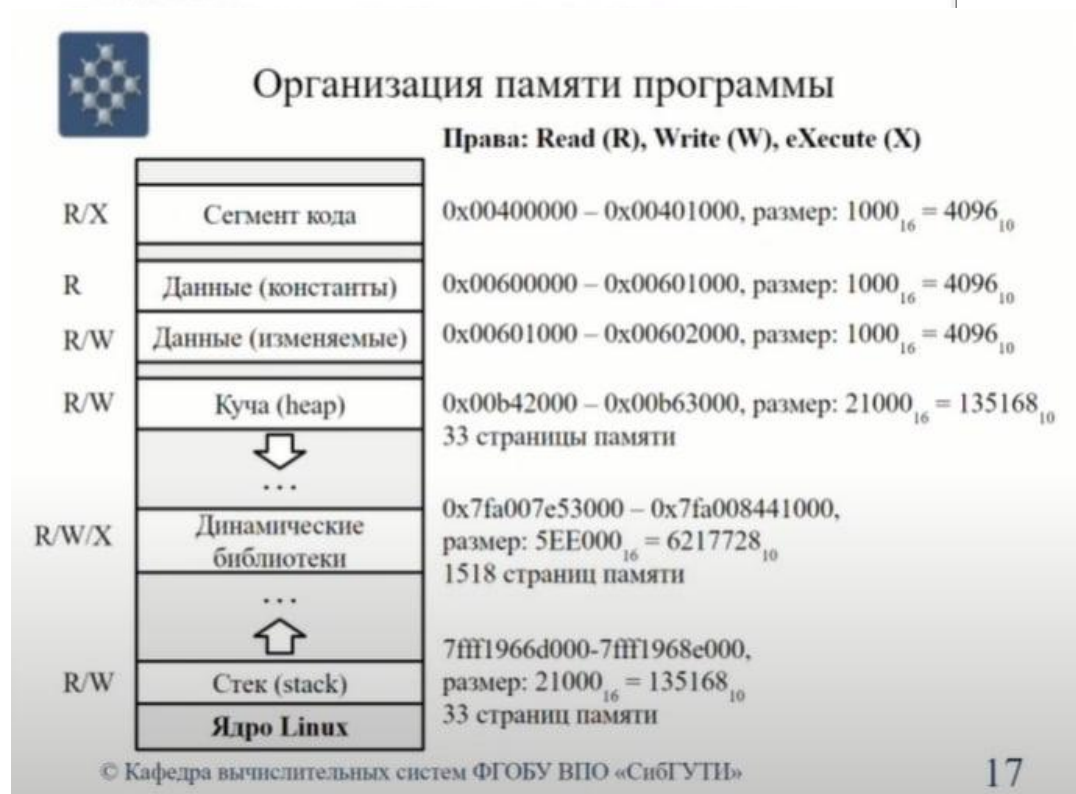
Динамические (они же разделяемые) библиотеки — еще один продукт компиляции с возможностью повторного использования. Как можно догадаться по названию, от статических библиотек они отличаются тем, что не входят в состав итогового исполняемого файла. Вместе этого их необходимо загружать и подключать во время запуска процесса.

Терминал 3.16. Создание динамической библиотеки из переносимых объектных файлов

```
$ gcc -shared 2d.o 3d.o trigon.o -o libgeometry.so
$ mkdir -p /opt/geometry
$ mv libgeometry.so /opt/geometry
$
```

Объектный файл содержит инструкции машинного уровня, эквивалентные единице трансляции. Однако они хранятся не в произвольном порядке, а сгруппированы в так называемые символы. Каждый переносимый объектный файл содержит только часть символов функций, необходимых для сборки полноценной исполняемой программы. Компоновщик собирает все символы из разных переносимых объектных файлов в один большой объектный файл, формируя тем самым исполняемую программу.

12. Организация памяти программы. Стек вызовов и принцип его работы.





Сегмент кода

Содержимое сегмента кода загружается из исполняемого файла, расположенного на носителе информации и содержит машинный код, сформированный компилятором.

Доступ: *чтение и исполнение.*

main: 0110110101111010111 101101010100110100010100 in: 1001110011100110101001 out: 01010101010011101111 calc: 10101011011001010111

prog
Функции: main: 0110110101111010111 101101010100110100010100 in: 1001110011100110101001 out: 01010101010011101111 calc: 10101011011001010111
Константы: path: "abcdefghijklnop" pi: 3.1415...
Инициализир. данные: errno: 0 global: 10 debug_level: 1000



Сегмент данных (констант)

Объявленные в программе константы размещаются в специальном регионе памяти, к которому возможен доступ *только чтение.*

main: 0110110101111010111 101101010100110100010100 in: 1001110011100110101001 out: 01010101010011101111 calc: 10101011011001010111
path: "abcdefghijklnop" pi: 3.1415...

prog
Функции: main: 0110110101111010111 101101010100110100010100 in: 1001110011100110101001 out: 01010101010011101111 calc: 10101011011001010111
Константы: path: "abcdefghijklnop" pi: 3.1415...
Инициализир. данные: errno: 0 global: 10 debug_level: 1000



Сегмент данных (модифицируемые)

Сегмент, содержащий статические и глобальные переменные, которые существуют на протяжении всего времени выполнения программы размещаются в отдельном сегменте.

Инициализирующие значения копируются из исполняемого файла.

Доступ: чтение и запись.

main: 0110110101111010111 101101010100110100010100 ...
path: "abcdefghijklmno"
pi: 3.1415...
errno: 0 global: 10 debug_level: 1000 global_2: 0

prog
Функции: main: 0110110101111010111 101101010100110100010100 in: 1001110011100110101001 out: 01010101010011101111 calc: 10101011011001010111
Константы: path: "abcdefghijklmno" pi: 3.1415...
Инициализир. данные: errno: 0 global: 10 debug_level: 1000



Регионы динамических библиотек

- Динамические библиотеки содержат как код так и данные, их загрузка осуществляется аналогично сегменту кода и данных соответственно.
- Дополнительной функцией данных сегментов является взаимная защита данных кучи от данных стека, которые растут навстречу друг другу. Это достигается за счет регионов, для которых запрещена запись.



Динамическое размещение программных объектов памяти

В процессе функционирования программы в ряде случаев возникает необходимость выделения памяти, которая не может быть предусмотрена заранее на этапе компиляции (статически), в частности:

1. Компилятор не может предугадать сколько раз и в какие моменты времени будет вызвана та или иная функция и будет ли она вызвана вообще. С другой стороны каждая функция имеет локальные переменные, которые необходимо где-то размещать. При этом после завершения функции эта область данных больше не нужна и может быть использована для других целей. Для обеспечения указанного функционала в программе используется *стек вызовов*.
2. При реализации многих программных архитектур возникает необходимость выделения памяти, не зависящей от той или иной функции. Время жизни такой памяти должно определяться явно программистом, а не тем, какие функции вызваны в данный момент. Такая память называется *динамической* и выделяется в области динамической памяти (куче, англ. heap).



Сегмент стека (стек)

Динамические библиотеки
...
Кадр (frame) №0
Кадр (frame) №1
Кадр (frame) №2
Кадр (frame) №3
Ядро Linux

Сегмент стека (call stack – стек вызовов) в теории вычислительных систем, LIFO-стек, хранящий информацию для:

1) возврата управления из вызванной подпрограммы в вызывающую подпрограмму;

2) возврата в подпрограмму из обработчика прерывания (в том числе при переключении задач в многозадачной среде).

Стек также используется для хранения локальных переменных каждой вызванной функции.

Для каждой вызываемой функции в стеке выделяется область памяти, называемая кадром (frame)

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 вернуть управление вызвавшей функции.



Пример работы стека

```

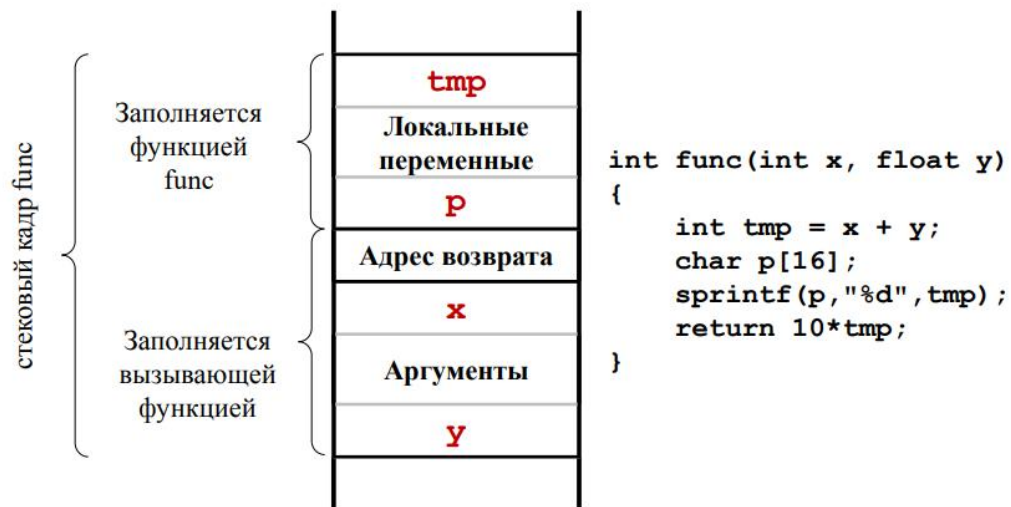
4 int func1(...){ ... }
3 int func2(...){ ... func1(...) ...}
2 int func3(...){ ... func2(...) ...}
1 int main(){ ... func3(...) ... }

```

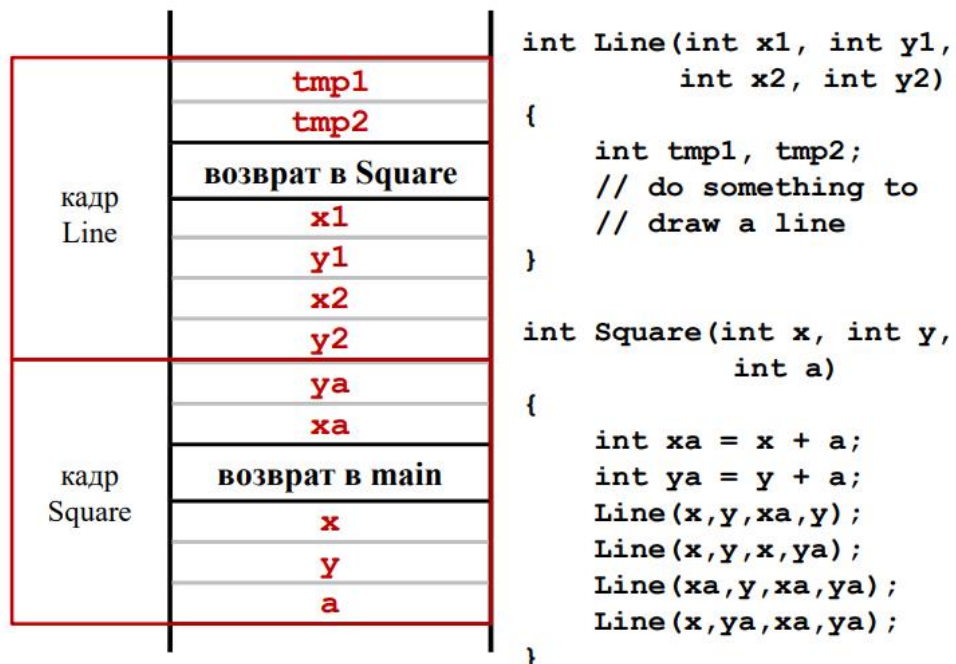
1	2	3	4
Динамические библиотеки	Динамические библиотеки	Динамические библиотеки	Динамч. библи.
...
	#0 func3()	#0 func2()	#0 func1()
#0 main()	#1 main()	#1 func3()	#1 func2()
Ядро Linux	Ядро Linux	#2 main()	#2 func3()
		Ядро Linux	#3 main()
			Ядро Linux



Стековый кадр



Стековый кадр (2)





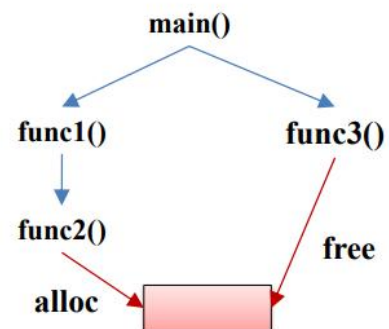
Стек и динамическая память

Структура данных стек хорошо подходит для хранения стековых кадров, т.к. до тех пор, пока текущая функция не закончит свою работу, ни одна из нижележащих функций не продолжит свое выполнение.

Динамическая память предназначена для хранения произвольных данных, время жизни которых заранее неизвестно.

Область памяти может быть выделена в одной функции и освобождена в другой.

Память продолжает существование даже тогда, когда выделившая ее функция извлекается из стека.



Динамическая память

Для реализации динамической памяти применяется структура данных "куча" (англ. heap). Куча запрашивает память у операционной системы. Эта память используется для размещения объектов, динамически созданных программой. Такие объекты продолжают свое существование до тех пор, пока не будут *явно освобождены* или программа не завершится.

В любой момент времени существования кучи вся динамическая память, разделена на *занятую* и *свободную*. Занятая память использована под размещение объектов, уже созданных и ещё не освобождённых к этому моменту времени. Из объёма свободной памяти *примитивы* работы с кучей могут выделять память под новые объекты.

Для хранения данных о принадлежности памяти к занятой или свободной обычно используется *служебная область памяти*.

Файл представляет собой непрерывную последовательность байт, каждый из которых может быть прочитано индивидуально.

Стандарт ANSI C предлагает два способа представления файла:

1. Текстовое представление
2. Двоичное представление

Программа на языке C открывает сразу 3 стандартных файла:

1. Стандартный ввод (`scanf`, `getchar`, `gets`)
2. Стандартный вывод (`printf`, `putchar`, `puts`)
3. Стандартный вывод ошибок

`FILE *fopen(const char *filename, const char *mode);`

где `filename` -- название файла, `mode` -- режим открытия.

Функция возвращает указатель на файл, если тот был успешно открыт. В противном случае -- `NULL`.

Режимы открытия файлов

`r` только чтение

`w` Только запись. Если **файл** существовал, то он переписывается.

`a` Добавление: открытие **файла** для записи в конец, или создание **файла**.

`r+` Открывает **файл** для обновления (чтение и запись).

`w+` Открывает **файл** для обновления (чтение и запись), переписывая **файл**, если он существует.

`a+` Открывает **файл** для записи в конец **файла** или для чтения.

Во втором параметре дополнительно может указываться символ `t` и `b` для указания текстовый файл или двоичный.

`rt, wt, at, rt+, wt+, at+`

`rt, wt, at, rt+, wt+, at+`

`int fclose(FILE *stream);`

где `stream` -- указатель на открытый файл.

Функция возвращает:

- 0 – файл успешно закрыт.
- 1 – произошла ошибка закрытия файла.

Чтение из двоичных файлов

```
size_t fread(void *buffer, size_t size, size_t num, FILE *stream);
```

Функция возвращает количество прочитанных блоков.

Если оно меньше num, то произошла ошибка или достигнут конец файла.

Запись в двоичный файл

```
size_t fwrite(const void *buffer, size_t size, size_t num, FILE *stream);
```

Функция возвращает количество записанных блоков.

Если оно меньше num, то произошла ошибка.

Чтение текущего смещения в файле

```
long int ftell(FILE *stream);
```

Изменение текущего смещения в файле

```
int fseek(FILE *stream, long int offset, int origin);
```

Значение origin может принимать три значения:

- SEEK_SET (0) – от начала файла.
- SEEK_CUR (1) – от текущей позиции.
- SEEK_END (2) – от конца файла.

Функция возвращает 0, если все нормально, и не ноль, если произошла ошибка.

```
int fscanf(FILE *stream, const char * format, [arg] ...);
```

Функция возвращает:

- больше 0 -- число успешно прочитанных переменных,
- 0 -- ни одна из переменных не была успешно прочитана,
- EOF -- ошибка или достигнут конец файла.

Форматированный вывод

```
int fprintf(FILE *stream, const char *format, [arg] ...);
```

Функция возвращает число записанных символов, если все нормально, и отрицательное значение, если произошла ошибка.

14. Битовые операторы. Логические побитовые операции и операции сдвига. Примеры.

Битовые операции — это тестирование, установка или сдвиг битов в байте или слове, которые соответствуют стандартным типам языка C `char` и `int`.

Битовые операторы не могут использоваться с `float`, `double`, `long double` и другими сложными типами.

Оператор	Действие
<code>&</code>	И
<code> </code>	ИЛИ
<code>^</code>	Исключающее ИЛИ
<code>~</code>	Дополнение
<code>>></code>	Сдвиг вправо
<code><<</code>	Сдвиг влево

	Битовое представление <code>x</code> после выполнения каждого оператора	Значение <code>x</code>
<code>char x;</code>		
<code>x = 7;</code>	00000111	7
<code>x = x << 1;</code>	00001110	14
<code>x = x << 3;</code>	01110000	112
<code>x = x << 2;</code>	11000000	192
<code>x = x >> 1;</code>	01100000	96
<code>x = x >> 2;</code>	00011000	24

Приоритет:

`~`

`>>/<<`

`&`

`^`

`|`

15. Динамическая память, примитивы работы с ней и принцип функционирования.

Динамическое распределение памяти – способ выделения оперативной памяти компьютера для объектов в программе в процессе ее исполнения:

- объекты размещаются в «куче» (англ. *heap*);
- при создании объекта указывается размер памяти;
- в случае успеха, выделенная область памяти «изымается» из кучи и недоступна при последующих операциях выделения памяти;
- память занятая ранее под какой-либо объект может быть освобождена;
- освобождаемая память возвращается в кучу и становится доступной при дальнейших операциях выделения памяти.

В языке Си существует четыре функции для динамического распределения памяти:

`malloc` (от англ. **m**emory **al**location, выделение памяти),
`calloc` (от англ. **c**lear **al**location, чистое выделение памяти)
`realloc` (от англ. **re**allocation, перераспределение памяти).
`free` (англ. **f**ree, освободить)

Функции `malloc`, `calloc`, `realloc` обеспечивают выделение памяти, функция `free` – освобождение памяти, возвращенной любой из функций ее выделения.

```
#include <stdlib.h>
void *malloc (size_t size);
void *calloc (size_t num, size_t size);
void *realloc(void *block, size_t size);
void *free(void *block);
```

Функция `malloc()` возвращает адрес на первый байт области памяти размером `size` байт, кото-рая была выделена из кучи. Если памяти недостаточно, чтобы удовлетворить запрос, функция `malloc()` возвращает нулевой указатель.

Функция `calloc()` работает также, как и `malloc()`, но инициализирует элементы нулевыми значениями.

Функция `realloc()` изменяет величину выделенной памяти, на которую указывает `ptr`, на новую величину, задаваемую параметром `newsize`. Величина `newsize` задается в байтах и может быть больше или меньше оригинала. Возвращается указатель на блок памяти, поскольку может возникнуть необходимость переместить блок при возрастании его размера. В таком случае содержимое старого блока копируется в новый блок и информация не теряется.

Динамическое выделение памяти характеризуется **существенными накладными расходами**, которые связаны с **поиском** свободного блока памяти, подходящего под запрос, т.е. размер которого близок к указанному в функции malloc.

Накладные расходы при перераспределении памяти могут быть существенно выше, так как помимо поиска подходящей области памяти необходимо также выполнить **копирование** содержимого из старой области в новую.

Для того, чтобы снизить указанные накладные расходы при организации динамически расширяемых массивов часто используют следующую стратегию:

1. Изначальный размер массива – x .
2. Если размера массива не достаточно для хранения данных – увеличить его размер вдвое: $x = 2 \cdot x$.

Таким образом, количество перераспределений логарифмически (а не линейно!) зависит от размера массива.