Конспект лекций по курсу «Программирование C/C++»

Ф.С. Пеплин

2024/2025 учебный год

Содержание

1 Компиляция. Целочисленные типы

1.1 Простейшая программа на Си

Напишем простейшую программу на Си, которая выводит на экран надпись `Hello, world` и больше ничего не делает (см. листинг 1).

Листинг 1: Наша первая программа

```
#include <stdio.h>

// This is one-line comment

/*
This is multi-line comment.
All this text is ignored by the compiler
*/

int main()
{
    printf("Hello, world!\n");
    return 0;
}
```

В строке 3 показан пример однострочного комментария: две косые черты и все, что следует за ними вплоть до конца строки, игнорируется. В строках 5 — 8 видим многострочный комментарий: все, что расположено между скобками /* и */, а также сами эти скобки, игнорируется и не влияет на работу программы. Комментарии нужны, чтобы описать для себя и коллег, что, как и зачем делает тот или иной фрагмент кода.

В строке 10 объявляется функция (подпрограмма) `main`. В языке `C` программа представляется в виде набора функций, но при этом только одна из них является главной и имеет название `main`. С нее начинается программа и в ней вызываются остальные функции. Текст функции заключен между фигурными скобками { и }. Ключевое слово `int` в строке 10 означает, что функция `main` возвращает целое

число. Целое число возвращается тому, кто вызвал функцию `main`, т.е. операционной системе, которая интерпретирует его как код завершения программы: 0 означает успешное завершение, иначе код ошибки. В строке 13 происходит возвращение нулевого кода завершения операционной системе. Отметим, что выполнение оператора `return` означает немедленное прекращение работы функции. Так, если бы мы поместили какие-то инструкции после 13 строки, то они не были бы выполнены.

Некоторые функции принимают аргументы из внешней среды, тогда они помещаются в круглые скобки после имени функции. Функция `main` в нашем примере ничего не принимает, поэтому в строке 10 скобки остались пустыми.

Вывод слов `Hello, world` осуществляется в строке 12, где вызывается функция `printf`. Данной функции передается один аргумент — строка для вывода на экран. Символы `\n` означают перевод строки.

Отметим, что вызываемая в 12 строке функция `printf` является библиотечной. Ее сигнатура (т.е. имя, тип возвращаемого значения, порядок и тип аргументов, но не конкретные выполняемые инструкции) находится в файле `stdio.h`, содержимое которого мы вставляем в нашу программу с помощью команды `#include` в строке 1.

1.2 Виды трансляторов

Программа в листинге 1 абсолютно не понятна компьютеру, который умеет оперировать лишь нулями и единицами. Для того, чтобы запустить программу на компьютере, ее нужно сперва перевести на понятный ему язык, т.е. на язык последовательностей нулей и единиц. Для этого нужна программа, которая называется **транслятором** (от английского translator — переводчик).

Трансляторы бывают двух типов: **компиляторы** и **интерпретаторы**.

Интерпретатор работает как переводчик-синхронист, переводя последовательно каждую строку исходного кода на язык компьютера. Пошаговое выполнение программы интерпретатором позволяет хо-

рошо локализовывать ошибки. К недостаткам такого подхода можно отнести то, что код программы получается слабо оптимизированным, поэтому программа работает медленно, занимает порядочно памяти и расходует много ресурсов процессора.

Языки программирования, для которых большинство трансляторов относятся к типу интерпретаторов, называются интерпретируемыми. Примерами таких языков являются `Python, Basic`.

Компилятор работает как литературный переводчик: он анализирует сразу весь текст программы, при переводе каждой строки учитывается контекст. Это позволяет компилятору лучше оптимизировать машинный код. Программа будет работать быстрее, занимая при этом меньше памяти. С другой стороны, т.к. компилятор преобразует сразу большие фрагменты кода, то иногда при наличии ошибок в коде их бывает сложнее выявить, то есть писать программы для компиляторов сложнее, чем для интерпретаторов.

Языки программирования, для которых большинство трансляторов относятся к типу компиляторов, называются компилируемыми. К ним относятся `C, C++, Go, Rust, Pascal`.

Для компиляции программы при использовании операционных систем `Mac OS X` и `Linux` можно использовать компилятор `gcc`.

Пусть текст программы сохранен в файле `main.c`. Для компиляции с использованием `gcc` нужно открыть в терминале папку, содержащую исходный код, и выполнить команду ($ означает приглашение к вводу в терминале, его не нужно вводить):

```
$ gcc main.c -o main
```

Далее программу можно запустить, набрав в терминале

```
$ ./main
```

При работе в среде `Windows` есть следующие варианты:

1. Использовать среду разработки `Visual Studio`.

   Скачать отсюда

   Как создать проект и запустить код

2. Установить `WSL` (`Windows Subsystem for Linux`) — тонкий слой виртуализации, который позволяет запускать приложения `Linux`,

не покидая привычного окружения `Windows`.

Для установки `WSL` нужно выполнить следующую команду в `PowerShell` (запуск от имени администратора):

```
wsl --install
```

После завершения установки запустите программу `Ubuntu` — это и есть терминал `Linux`.

1.3 Этапы компиляции

Этап 1. Препроцессор.

На данном этапе происходит предварительная обработка исходного текста программы. Препроцессор ищет в коде программы т.н. директивы препроцессора — команды, начинающиеся с #, и выполняет предписываемые директивами манипуляции с исходным текстом. В листинге 1 имеется лишь одна директива `#include`, которая командует препроцессору скопировать на ее место текст из указанного файла (в нашем случае это файл `stdio.h`).

Кроме того, препроцессор также удаляет комментарии из исходного года.

Посмотреть на результат работы препроцессора можно с помощью флага `-E`:

```
$ gcc -E hello.c -o hello.i
```

Этап 2. Компиляция.

На данном этапе полученный в результате работы препроцессора исходный код программы переводится на язык ассемблера — удобную для человека запись машинного кода. См. листинг 2.

```
$ gcc -S hello.i -o hello.s
```

Листинг 2: Файл hello.s

```
.file   "main.c"
.text
.section        .rodata
.LC0:
.string "Hello, world!"
```

```
.text
.globl  main
.type   main, @function
main:
.LFB0:
.cfi_startproc
endbr64
pushq   %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq    %rsp, %rbp
.cfi_def_cfa_register 6
leaq    .LC0(%rip), %rax
movq    %rax, %rdi
call    puts@PLT
movl    $0, %eax
popq    %rbp
.cfi_def_cfa 7, 8
ret
.cfi_endproc
.LFE0:
.size   main, .-main
.ident  "GCC: (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0"
.section        .note.GNU-stack,"",@progbits
.section        .note.gnu.property,"a"
.align 8
.long   1f - 0f
.long   4f - 1f
.long   5
0:
.string "GNU"
1:
.align 8
.long   0xc0000002
.long   3f - 2f
```

```
2:
.long    0x3
3:
.align 8
4:
```

Этап 3. Ассемблирование.

На этом этапе берется ассемблерный код, полученный на стадии компиляции, и преобразуется в машинный код (нули и единицы).

```
$ gcc -c hello.s -o hello.o
```

Этап 4. Линковка.

Исходный текст реальный проектов обычно занимает сотни тысяч и миллионы строк, которые расположены в нескольких файлах. Каждый из этих файлов подвергается трем описанным выше этапам, после выполнения которых получаем несколько объектных файлов `file1.o, file2.o, ..., filen.o`. На этапе линковки происходит стыковка всех этих файлов в единый исполняемый файл.

```
$ gcc hello.o -o hello
```

В нашем случае исходный код содержится в одном файле. Однако мы вызывали функцию `printf` из стандартной библиотеки. Поэтому на этапе линковки файл `hello.o` стыкуется с библиотечным бинарным файлом `stdio.o`.

1.4 Целочисленные типы данных

В языке `C` числа, как и любые данные, хранятся в переменных. Переменная — это именованная область оперативной памяти. Каждая переменная имеет свой тип, который не меняется на протяжении всей программы.

В языке `C` имеется два основных типа целых чисел — `char` и `int`. Путем добавления слов `signed, unsigned, long, short` можно получать много разных целочисленных типов. Итого получаем типы `char`, `short int`, `int`, `long int`, `long long int`, каждый из которых может быть как `signed`, так и `unsigned`.

Отметим, что `signed int` и `int` — это один и тот же тип, т.е. пе-

ременная типа `int` всегда знаковая. В то же время тип `char` может быть на одной системе знаковым, а на другой беззнаковым. Поэтому, если хотим хранить отрицательные числа в `char`, то нужен тип `signed char`.

Примеры объявлений целочисленных переменных:

```
signed int x = 2;
unsigned long long int Var123;
long int qwerty;
int x;
int y = 2;
int a, b, c = 4;
int d = y + c; // d равно 6
```

Последняя команда — это оператор присваивания, после выполнения которого сумма `y + c` окажется в области памяти, закрепленной за переменной `d`.

Имена переменных чувствительны к регистру. Так, переменные именами `cat` и `Cat` являются различными.

Если объявленная переменная не инициализирована (ей не присвоено значение), то в ней может находиться что угодно (мусор). Ее использовать в арифметических операциях нельзя.

Перед типом переменной можно поставить ключевое слово `const`, что означает, что данную переменную нельзя менять. Например, выполнение следующего кода приведет к ошибке компиляции

```
const x = 2;
x = 3;
```

Чтобы знать, какие числа можно хранить в той или иной целочисленной переменной, нужно знать ее размер. Стандарт языка Си не регламентирует конкретных размеров, однако гарантирует, что размер `char` $\leq$ размер `short int` $\leq$ размер `int` $\leq$ размер `long int` $\leq$ размер `long long int`.

Возникает вопрос: как узнать размер целочисленных типов на конкретной системе? Для этого можно воспользоваться функцией

`sizeof`, которая возвращает количество занимаемых переменной байтов, а также константами из файла `limits.h`. Оба способа проиллюстрированы в листинге 3. Типичные значения приведены на рисунке 1.

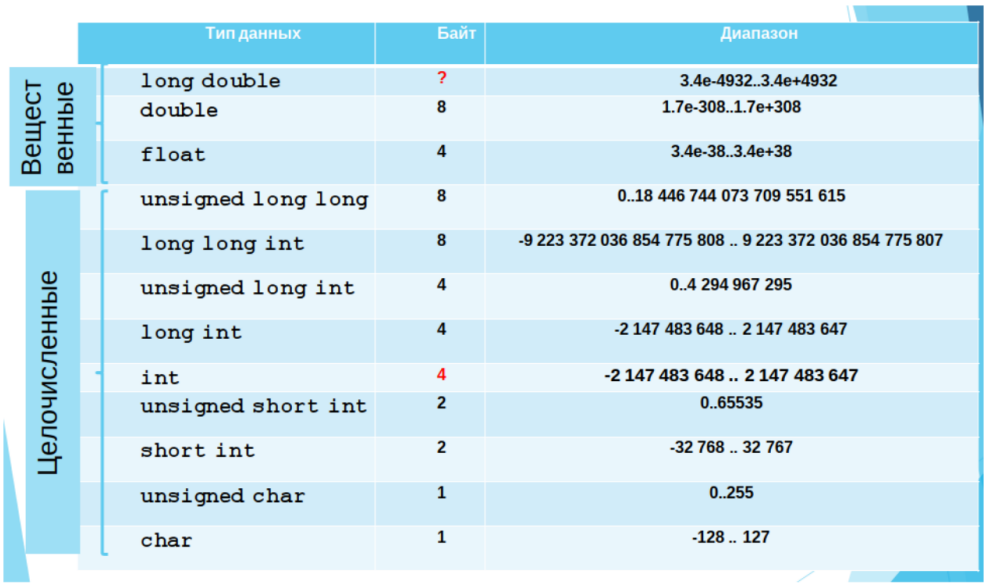
Отметим, что `sizeof(char)` всегда равно 1, даже если переменные данного типа занимают не 8 бит, а больше (меньше нельзя по стандарту). На таких системах байт равен не 8 битам, а размеру (в битах) переменных типа `char`. Размеры остальных целочисленных переменных кратны размеру `char`.

Листинг 3: Определение размера целочисленных переменных

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

int main()
{
    printf("Number of bits in char: %d\n", CHAR_BIT);
    printf("Minimal value of char: %d\n", CHAR_MIN);
    printf("Maximum value of char: %d\n", CHAR_MAX);
    printf("Minimal value of signed char: %d\n", SCHAR_MIN);
    printf("Maximal value of signed char: %d\n", SCHAR_MAX);
    printf("Maximal value of unsigned char: %u\n", UCHAR_MAX);
    printf("Minimal value of short int: %d\n", SHRT_MIN);
    printf("Minimal value of short int: %d\n", SHRT_MAX);
    printf("Maximal value of unsigned short int: %u\n", USHRT_M
    printf("Number of bits in int: %ld\n", sizeof(int) * CHAR_
    printf("Minimal value of int: %d\n", INT_MIN);
    printf("Maximal value of int: %d\n", INT_MAX);
    printf("Maximal value of unsigned int: %u\n", UINT_MAX);
    printf("Minimal value of long int: %ld\n", LONG_MIN);
    printf("Maximal value of long int: %ld\n", LONG_MAX);
    printf("Maximal value of unsigned long int: %lu\n", ULONG_M
    return 0;
}
```

Попробуем понять, почему диапазоны на рис. 1 именно такие.



	Тип данных	Байт	Диапазон
Вещественные	long double	?	3.4e-4932..3.4e+4932
	double	8	1.7e-308..1.7e+308
	float	4	3.4e-38..3.4e+38
Целочисленные	unsigned long long	8	0..18 446 744 073 709 551 615
	long long int	8	-9 223 372 036 854 775 808 .. 9 223 372 036 854 775 807
	unsigned long int	4	0..4 294 967 295
	long int	4	-2 147 483 648 .. 2 147 483 647
	int	4	-2 147 483 648 .. 2 147 483 647
	unsigned short int	2	0..65535
	short int	2	-32 768 .. 32 767
	unsigned char	1	0..255
	char	1	-128 .. 127

Рис. 1: Типичные размеры целочисленных типов

Для этого рассмотрим воображаемый четырехбитовый тип `half_char`. Перечислим все значения, которые может принимать переменная этого типа, как знаковая, так и беззнаковая (таблица 1.4). Для знаковой переменной считаем, что старший бит отвечает за знак: 0 соответствует положительному числу, 1 отрицательному. Обратим внимание, что сумма положительного и обратного к нему отрицательного числа в четырехбитовой арифметике равна 0, например $5 + (-5) = 0101 + 1011 = 10000 = 0000$.

x_{10}	x_{10}	x_2
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	-8	1000
9	-7	1001
10	-6	1010
11	-5	1011
12	-4	1100
13	-3	1101
14	-2	1110
15	-1	1111

Таблица 1: Диапазон значений целочисленной четырехбитовой переменной

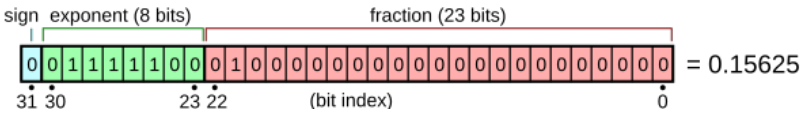


Рис. 2: Тип float

2 Дробные числа. Операторы и выражения

2.1 Представление дробных чисел. Погрешность операций с действительными числами

Любое действительное число можно представить в экспоненциальном виде, например,

$$12345 = \underbrace{1.2345}_{\text{Мантисса}} \times 10^{\overbrace{4}^{\text{порядок}}} \tag{1}$$

В двоичной записи представление (1) запишется в виде

$$12345 = 1.\underbrace{1000000111001}_{\text{Мантисса}} \times 2^{\overbrace{13}^{\text{порядок}}} \tag{2}$$

Мантисса также называется significand и fraction. Порядок еще называют exponent.

В языке `C` дробные типы обозначаются `float` и `double`. Несмотря на то, что стандарт языка не регламентирует конкретные размеры типов данных, в большинстве случаем `float` соответствует типу `float32` (см. рис. 2), а `double` — типу `float64` (см. рис. 3) из стандарта IEEE 754.

Естественно возникают два вопроса:

1. Какие максимальные и минимальные значения можно хранить в типах `float` и `double`?

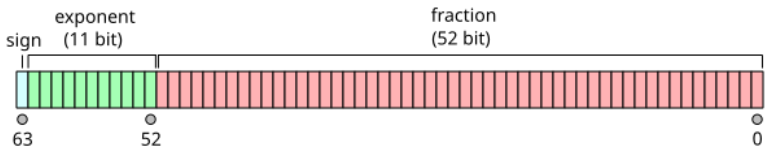


Рис. 3: Тип double

2. Какая точность представления дробного числа при работе с типами `float` и `double`?

Дадим ответы на эти вопросы на примере 32-битного `float`.

Порядок (экспонента) занимает 8 бит, которыми можно закодировать $2^8 = 256$ различных состояний, т.е. числа от 0 до 255. Экспонента всегда считается положительной, а из нее вычитается смещение, которое для 8-битной экспоненты равно 127 ($2^{8-1} - 1$). Числа 0 и 255 зарезервированы: 255 обозначает бесконечность (если мантисса равна нулю) или `NaN`[1] (если мантисса нуль), а число 0 позволяет кодировать субнормальные числа[2]. Таким образом, экспонента получается от 1 до 254, а вычитая сдвиг 127, получаем от -126 до 127. Учитывая, что мантисса кодируется 23 битами, получаем, что число перед степенью двойки меняется от 1 до $2 - 2^{-23}$. Таким образом, минимальное положительное значение, которое хранит `float32`, составляет $2^{-126} \approx 10^{-38}$, а максимальное $(2 - 2^{-23}) \times 2^{127} \approx 10^{38}$. Относительная погрешность вычислений составляет $2^{-23} \approx 10^{-7}$.

Рассуждая аналогично для `float64`, можно заключить, что относительная погрешность представления $2^{-52} \approx 10^{-15}$. Минимальное по модулю значение 2^{-1022}, максимальное — $(2-2^{-52}) \times 2^{2023} \approx 10^{308}$.

Возникновение погрешности при работе с действительными числами показано в листинге 4. Ошибка возникает уже в третьем знаке после запятой. Это связано с тем, что при работе с действительными числами реализуется относительная погрешность, а не абсолютная:

[1]Not a number — не числовое значение.

[2]Субнормальным называется число, для которого экспонента равна нулю, а мантисса отлична от нуля. Тогда считается, что такое число для `float32` кодируется в виде $(-1)^s \times 0.xxxxxx \times 2^{-126}$. Субнормальные числа сглаживают скачок между минимальным числом и нулем.

при работе с большими числами с 15 нулями относительная погрешность 10^{-15} приводит к абсолютной погрешности порядка единиц.

Листинг 4: Округление при выполнении операций с действительными числами

```
#include <stdio.h>

int main()
{
    double x = 123456789999999.11;
    double y = 123456789999998.1;
    printf("%lf\n", x - y); // 1.015625
    return 0;
}
```

2.2 Операторы

2.2.1 Арифметические операции, приведение типов

Операторы сложения, вычитания, умножения и деления ведут себя в целом предсказуемо.

По логике Си результат операций между целыми числами — целое число, в том числе деление целых чисел тоже целое число. Например, при попытке присвоить дробной переменной значение 5/6 ответ будет 0, так как целая часть деления это 0. Чтобы ответ стал дробным, нужно привести хотя бы один из операндов к дробному типу `((double)5)/6`, либо просто `5.0/6`. В обоих случаях делитель сам преобразуется к дробному типу.

Помимо целочисленного деления, есть также операция вычисления остатка от деления %.

Вообще, если в арифметическом выражении фигурируют операнды разных числовых типов, то более «узкий» тип приводится к более «широкому», и результат операции также относится к «широкому» типу.

Если все операнды знаковые, то действуют следующие правила приведения типов:

Если один из операндов `long double`, то второй привести к типу `long double`.

Иначе, если один из операндов `double`, то второй привести к типу `double`.

Иначе, если один из них `float`, то второй привести к `float`.

Иначе, `char` и `short int` привести к типу `int`.

Далее, если один `long`, то второй привести к `long`.

Если один из операндов без знака, а другой со знаком, и размеры переменных равны, то беззнаковый тип почему-то считается более «широким», см. листинг 5.

Листинг 5: Конверсия знаквых и беззнаковых чисел

```
#include <stdio.h>

int main()
{
        unsigned int ui = 1;
        long int li = -1;
        int i = -1;
        printf("%d\n", li < ui); // 1 - true
        printf("%d\n", i < ui);  // 0 - false
        return 0;
}
```

Инкремент.

Увеличение переменной `a` на значение `b`:

```
a = a + b;
a += b;
```

Увеличение на единицу:

```
a = a + 1;
a += 1;
a++;
++a;
```

(аналогично уменьшение на единицу)

Отличие двух последних форм инкремента / декремента заключается в том, что если ++ (или - -) расположен справа, то значением выражения будет неинкрементированное значение переменной, см. листинг 6.

Листинг 6: Демонстрация инкремента

```
#include <stdio.h>

int main()
{
    int x = 1;
    printf("%d\n", x++); // 1
    printf("%d\n", ++x); // 3
    printf("%d\n", (x++) * (++x)); // ub!
    return 0;
}
```

2.2.2 Логические операторы

Истина — любое ненулевое число. Ложь — это только нуль.

Конъюнкция. `a && b` истинно тогда и только тогда, когда истинны оба операнда.

Дизъюнкция. `a || b` истинно тогда и только тогда, когда истинен хотя бы один из операндов (или оба).

Отрицание. `!a` истинно тогда и только тогда, когда `a` ложь.

Листинг 7: Демонстрация конъюнкции

```
#include <stdio.h>

int main()
{
    int a = 10, b = 20;

    if (a > 0 && b > 0) {
        printf("Both values are greater than 0\n");
```

```
    }
    else {
        printf("Both values are less than 0\n");
    }
    return 0;
}
```

Листинг 8: Демонстрация конъюнкции

```
#include <stdio.h>

int main()
{
    int a = 10, b = 20;

    if (a > 0 && b > 0) {
        printf("Both values are greater than 0\n");
    }
    else {
        printf("Both values are less than 0\n");
    }
    return 0;
}
```

Листинг 9: Демонстрация дизъюнкции

```
#include <stdio.h>

int main()
{
    int a = -1, b = 20;

    if (a > 0 || b > 0) {
        printf("Any one of the given value is "
        "greater than 0\n");
    }
    else {
```

```
        printf("Both values are less than 0\n");
    }
    return 0;
}
```

Листинг 10: Демонстрация отрицания

```
#include <stdio.h>

int main()
{
    int a = 10, b = 20;

    if (!(a > 0 && b > 0)) {
        printf("Both values are greater than 0\n");
    }
    else {
        printf("Both values are less than 0\n");
    }
    return 0;
}
```

Замечание. Конъюнкция и дизъюнкции выполняются слева направо. Если первый аргумент конъюнкции ложь (или первый аргумент дизъюнкции истина), то проверять второй аргумент бессмысленно, ибо значение всего выражения и так понятно. Поэтому имеет смысл более ресурсоемкие выражения ставить вторым (третьим и т.д.) операндом.

Операторы сравнения. `a == b, a != b, a < b, a <= b, a > b, a >= b`.

2.2.3 Битовые операции

Побитовая конъюнкция. Оператор &.

Побитовая дизъюнкция. Оператор |.

Побитовое исключающее или (XOR). Оператор ^. Сравнивает попарно каждые два бита и ставит единичку, если биты разные.

Сдвиг влево. Оператор `a « n`. Осуществляет циклический сдвиг `a` влево на `n` позиций.

Сдвиг вправо. Оператор `a » n`. Осуществляет циклический сдвиг `a` вправо на `n` позиций.

Побитовое отрицание. Оператор $\tilde{}a$. Осуществляет побитовую инверсию a.

```
#include <stdio.h>
int main()
{
    // a = 5 (00000101 in 8-bit binary), b = 9 (00001001 in
    // 8-bit binary)
    unsigned int a = 5, b = 9;

    // The result is 00000001
    printf("a = %u, b = %u\n", a, b);
    printf("a&b = %u\n", a & b);

    // The result is 00001101
    printf("a|b = %u\n", a | b);

    // The result is 00001100
    printf("a^b = %u\n", a ^ b);

    // The result is 11111111111111111111111111111010
    // (assuming 32-bit unsigned int)
    printf("~a = %u\n", a = ~a);

    // The result is 00010010
    printf("b<<1 = %u\n", b << 1);

    // The result is 00000100
    printf("b>>1 = %u\n", b >> 1);

    return 0;
```

}

3 Условия и циклы

3.1 Условный оператор

3.1.1 Оператор `if-else`

Условный оператор позволяет выполнять те или иные инструкции в зависимости от того, выполнено или нет условие:

```
if(condition)
{
    //do if condition is true
}
else
{
    //do if condition is false
}
```

Если требуется проверить несколько условий, то допустима конструкция:

```
if(condition1)
{
    //do if condition1 is true
}
else if(condition2)
{
    //do if condition2 is true
}
else if(condition3)
{
    //do if condition3 is true
}
else
{
    //do if all conditions are false
}
```

Если при невыполнении условия ничего делать не требуется, то `else` можно опустить:

```
if(condition)
{
    //do if condition is true
}
```

Если требуется выполнить только одну инструкцию, то операторные скобки { и } можно опустить.

3.1.2 Тернарный оператор

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

Синтаксис тернарной операции в языке C выглядит следующим образом:

Условие ? Выражение1 : Выражение2;

Если выполняется условие, то тернарная операция возвращает выражение1, в противном случае — выражение2.

Тернарные операции, как и операции условия, могут быть вложенными. Для разделения вложенных операций используются круглые скобки.

Если при невыполнении условия не нужно ничего выполнять, то двоеточие и Выражение2 можно опустить.

3.1.3 Оператор `switch`

```
case(IntegralValue)
{
    case A:
        // do if IntegralValue == A
        break;
    case B:
        // do if IntegralValue == B
```

```
        break;
    default:
        // do if neither cases A, B, etc and others are suitabl
        break;
}
```

3.2 Циклы

Циклы позволяют повторять одни и те же команды в зависимости от того, выполнено некоторое условие или нет. Если выполняемая команда всего одна, то операторные скобки { и } можно опустить.

3.2.1 Цикл `while`

```
while(condition)
{
    // repeat while condition is met
}
```

3.2.2 Цикл `do-while`

```
    do
    {
        // repeat while condition is met
    }
    while(condition)
```

Отличие от цикла `while` заключается в том, что действия будут выполнены хотя бы один раз вне зависимости от истинности условия (потому что условие проверяется после первой итерации).

3.2.3 Цикл `for`

```
for(init; cond; incr)
{
    // repeat while condition is met
}
```

Цикл `for` работает следующим образом. Перед первой итерацией выполняются команды секции `init`. Как правило, это инициализация переменной-счетчика. Вторая секция — это условие. Если условие истинно, то выполняется тело цикла. После каждой итерации цикла выполняется команда в третьей секции `incr`.

3.3 Примеры

3.3.1 Пример 1 (калькулятор)

```
#include <stdio.h>

int main()
{
    char op;
    int a, b, res;
    printf("Enter operation (+, -, /, *): ");
    scanf("%c", &op);
    printf("Enter two operands: ");
    scanf("%d %d", &a, &b);
    if (op == '+')
        res = a + b;
    else if (op == '-')
        res = a - b;
    else if (op == '*')
        res = a * b;
    else if(op == '/')
        res = a / b;
    else
    {
```

```
            printf("You entered wrong operation!\n");
            return 0;
        }
        printf("Res=%d\n", res);

        return 0;
}
```

3.3.2 Пример 2 (количество корней квадратного уравнения)

```
#include <stdio.h>

int main()
{
        printf("Enter a, b, c: ");
        double a,b,c;
        scanf("%lf %lf %lf", &a, &b, &c);
        printf("The number of roots is %d\n",
            b*b-4*a*c > 0 ? 2 : (b*b-4*a*c == 0 ? 1 : 0));

        return 0;
}
```

3.3.3 Пример 3 (алгоритм Евклида)

```
#include <stdio.h>

int main()
{
        int a, b;
        scanf("%d %d", &a, &b);
        while (a > 0 && b > 0)
        {
```

```
        if (a > b)
        a %= b;
        else
        b %= a;
    }

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

    return 0;
}
```

3.3.4 Пример 4 (вычисление числа π)

$$\pi = 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots\right)$$

```
#include <stdio.h>

int main()
{
    double pi = 0;
    int n;
    scanf("%d", &n);

    double factor = 1.0;
    for (int i = 0; i < n; i++, factor *= -1)
        pi += factor / (2 * i + 1);

    pi*=4;
    printf("pi=%lf\n", pi);

    return 0;
}
```

3.3.5 Пример 5 (метод Ньютона)

Рассмотрим решение уравнения

$$f(x) = 0$$

В качестве конкретного примера возьмем

$$\sin x - \exp x = 0$$

Итеративная процедура метода Ньютона (или метода касательных):

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

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

#define EPS 1e-5

int main()
{

    double x = 1; // initial guess
    while (fabs(sin(x) - exp(x)) > EPS)
    {
        x -= (sin(x) - exp(x)) / (cos(x) - exp(x));
    }

    printf("x=%lf\n", x);
    printf("f=%lf\n", (sin(x) - exp(x)));
    return 0;
}
```

3.3.6 Пример 6 (анализатор строки)

```c
#include <stdio.h>

int main()
{
    int nwhite = 0;
    int nother = 0;
    int ndigits[10];
    for (int i = 0; i < 10; i++)
    ndigits[i] = 0;
    char ch;
    while ((ch = getchar()) != EOF)
    {
        switch (ch)
        {
            case '0': case '1': case '2':
            case '3': case '4': case '5':
            case '6': case '7': case '8': case '9':
                ndigits[ch - '0']++;
                break;
            case ' ':
            case '\n':
            case '\t':
                nwhite++;
                break;
            default:
                nother++;
                break;
        }
    }
    printf("\n Digits\n");
    for(int i=0; i<10; i++)
        printf("%d %d\n", i, ndigits[i]);
    printf("White space = %d\n", nwhite);
    printf("Other = %d\n", nother);
    return 0;
```

```
}
```

4 Иерархия памяти

4.1 Статические и глобальные переменные

Область видимости переменной — это часть программы, в которой возможно обращение к переменной. Область видимости обозначается открывающей и закрывающей фигурными скобками { и }. Все переменные, объявленные внутри блока, видны только в нем. См. листинг 11.

Листинг 11: Область видимости

```
#include <stdio.h>

int main()
{
  int x = 1;
  printf("%d\n", x); // 1
  {
    int x = 2;
    printf("%d\n", x); // 2
  }
  printf("%d\n", x); // 1
  return 0;
}
```

Статическая переменная — это переменная, которая не теряет своего значения при выходе из области видимости. Статическая переменная не инициализируется повторно при заходе в тот же блок кода. В отличие от глобальной переменной, статическая переменная видна лишь внутри того блока, где она была объявлена.

Отличие статических переменных от обычных иллюстрируется программов в листинге 12. Первый цикл выведет последовательность из пяти единиц, т.к. при каждом заходе в области видимости первого цикла происходит повторная инициализация переменной `x`. Второй цикл выведет последовательность чисел от 1 до 5, потому что статическая переменная инициализируется только один раз и при выходе

из блока кода ее значение не теряется, сохраняясь также при повторной инициализации.

Листинг 12: Статические переменные

```
#include <stdio.h>

int main()
{
  for (int i = 0; i < 5; i++)
  {
    int x = 1;
    printf("%d\n", x++);
  }
  for (int i = 0; i < 5; i++)
  {
    static int x = 1;
    printf("%d\n", x++);
  }
  return 0;
}
```

4.2 Иерархия памяти

Выделяют следующие уровни памяти, занимаемой программой:

1. Текстовый сегмент, в котором хранятся инструкции программы.

2. Сегмент данных, в котором хранятся глобальные и статические переменные.

   (a) Сегмент инициализированных данных, в нем хранятся статические и глобальные переменные, которые были проинициализированы программистам (им присвоено начальное значение при объявлении).

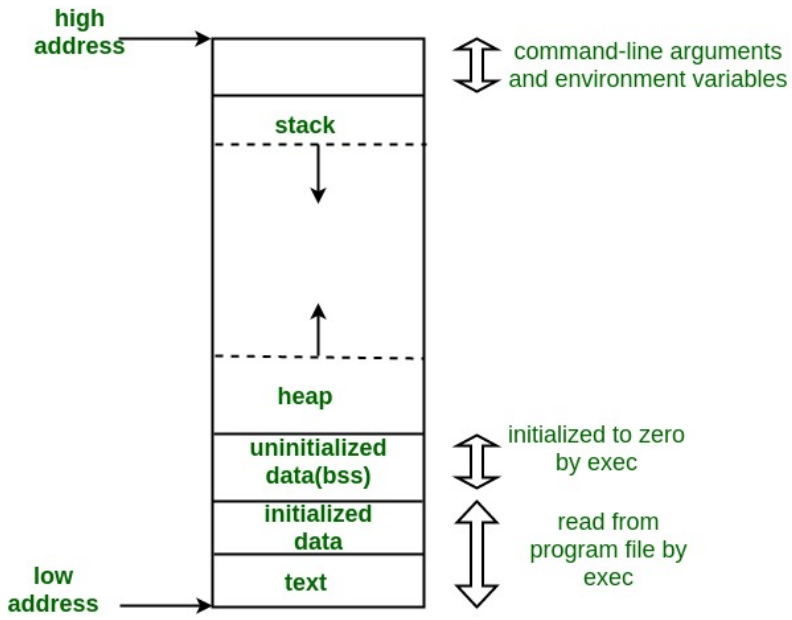


Рис. 4: Иерархия памяти, занимаемой программой

i. Подсегмент, допускающий чтение и запись.

ii. Подсегмент, допускающий только чтение.

(b) Сегмент неинициализированных данных (bss), содержащий глобальные и статические переменные, которые не были проинициализированы при объявлении. Их значение автоматически считается равным 0.

3. Стек. Переменные, объявленные в какой-нибудь функции (например, `main`). Размер стека ограничен несколькими мегабайтами.

4. Куча (heap). Эта память выделяется явным образом программистом с помощью функций `malloc, calloc, realloc`.

Узнать размер текстового сегмента и сегментов данных скомпилированной программы можно с помощью `Unix`-команды `size`, которая в качестве параметра получает имя исполняемого файла. Если работаете под `Windows`, то ~~снесите ее к чертям и поставьте `Ubuntu`~~ используйте `WSL`.

5 Массивы

5.1 Массив на стеке

Массив — это совокупность переменных одного типа, которые хранятся в памяти последовательно одна за другой, и к которым возможен единообразный доступ по общему имени массива.

Пример работы с массивом:

Листинг 13: Массив на стеке

```
#include <stdio.h>
#define SIZE 10

int main()
{
  int a[SIZE];
  for(int i=0; i<SIZE; i++)
    scanf("%d", &a[i]);
  for(int i=0; i<SIZE; i++)
    printf("%d ", a[i]);
  printf("\n");
  return 0;
}
```

Длина массива может определяться во время работы программы (`Variable Length Array`):

Листинг 14: VLA-массив на стеке

```
#include <stdio.h>

int main()
{
  int n;
  scanf("%d", &n);
  int a[n];
  for(int i=0; i<n; i++)
```

```
    scanf("%d", &a[i]);
  for(int i=0; i<n; i++)
    printf("%d ", a[i]);
  printf("\n");
  return 0;
}
```

Отметим, что размер стека ограничен несколькими мегабайтами, поэтому создать большой массив на стеке не получится (вероятнее всего, будет ошибка `Segmentation fault`):

Листинг 15: Стек слишком мал для такого массива

```
#include <stdio.h>
#define SIZE 10000000

int main()
{
  int a[SIZE];
  return 0;
}
```

5.2 Глобальный массив

Зачем нужно объявлять массив глобальным (вне какой-либо функции)?

1. Чтобы массив было видно не только из функции `main`, но и из других функций.

2. Чтобы массив разместить в сегменте инициализированных данных, а не на стеке: второй сильно меньше по размеру, чем первый (см. листинг 16).

Листинг 16: Глобальный массив ОК

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

```
#define SIZE 10000000

int a[SIZE];

int main()
{
  return 0;
}
```

5.3 Динамический массив (массив на куче)

Глобальный массив подходит, когда нужно разместить большое количество данных, для которого на стеке не хватает места. Но он расширяет область видимости переменной (что может быть не нужным). Более того, размер глобального массива должен быть известен во время компиляции (`VLA` подходит только для локальных массивов).

Если нужно разместить большой массив, размер которого определяется во время работы программы, нужно использовать массив на куче. Память для массива на куче выделяется с помощью функции `malloc` или `calloc`, а освобождается с помощью `free()`.

Листинг 17: Использование `malloc`

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

int main()
{
  int n;
  scanf("%d", &n);
  int *a = malloc(n * sizeof(int));
  for (int i = 0; i < n; i++)
    scanf("%d", &a[i]);
  for (int i = 0; i < n; i++)
    printf("%d ", a[i]);
```

```
    printf("\n");
    free(a);
    return 0;
}
```

Листинг 18: Использование `calloc`

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

int main()
{
    int n;
    scanf("%d", &n);
    int *a = calloc(n, sizeof(int));
    for (int i = 0; i < n; i++)
        scanf("%d", &a[i]);
    for (int i = 0; i < n; i++)
        printf("%d ", a[i]);
    printf("\n");
    free(a);
    return 0;
}
```

Функцйя `calloc`, в отличие от `malloc`, инициализирует элементы массива нулями. Элементы массива, память для которого выделяется функцией `malloc`, по умолчанию содержат произвольные значения.

Листинг 19: О важности `free()`

```
1 #include <stdio.h>
2 #include <math.h>
3 #include <stdlib.h>
4
5 int main()
6 {
7
```

```
 8       unsigned int n = 4000000000;
 9       int *a;
10
11       while (1)
12       {
13           a = malloc(n * sizeof(int));
14           if (a == 0)
15           {
16               perror("Error: ");
17               break;
18           }
19           free(a);
20       }
21       return 0;
22   }
```

Функция `realloc` используется, чтобы изменить размер уже выделенной для массива памяти.

Листинг 20: Использование `realloc()`

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 int main()
 5 {
 6       int n;
 7       scanf("%d", &n);
 8       int *a = calloc(n, sizeof(int));
 9       a = realloc(a, (n + 2) * sizeof(int));
10       for (int i = 0; i < n + 2; i++)
11           scanf("%d", &a[i]);
12       for (int i = 0; i < n + 2; i++)
13           printf("%d ", a[i]);
14       printf("\n");
15       free(a);
16       return 0;
```

```
17 }
```

5.4 Стек (структура данных)

Слово «стек» имеет и другое значение, помимо области памяти программы. Это также структура данных которая работает по принципу `LIFO (Last In First Out)`. Стек поддерживает такие операции, как `PUSH` (поместить в конец стека), `POP` (взять последний добавленный элемент и удалить его), `TOP` (взять последний добавленный элемент без его удаления).

Реализация стека показана в листинге 21. Использовано ключевое слово `enum`, которое будет рассматриваться далее. Здесь лишь скажем, что данное слово задает перечисление — новый тип, который может принимать лишь значения, перечисленные в фигурных скобках.

Листинг 21: Стек

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

#define MAX_STACK_SIZE 5
enum STACK_OPERATION {
   PUSH,
   POP,
   TOP,
   IS_EMPTY,
   EXIT
};

void menu();

int main()
{
   int stack[MAX_STACK_SIZE] = { 0 };
```

```
    int top = 0;
    enum STACK_OPERATION operation;
    while (1)
    {
        menu();
        scanf("%d", &operation);
        switch (operation)
        {
            case PUSH:
            if (MAX_STACK_SIZE == top)
            printf("Stack is full\n");
            else
            scanf("%d", &stack[top++]);
            break;
            case POP:
            if (!top)
            printf("Stack is empty\n");
            else
            --top;
            break;
            case TOP:
            if (!top)
            printf("Stack is empty\n");
            else
            printf("Top is %d\n", stack[top - 1]);
            break;
            case IS_EMPTY:
            printf("Stack is %sempty\n", top ? "not " : "");
            break;
            case EXIT:
            return 0;
        }
    }
}
```

```
void menu()
{
   printf("0 - push\n");
   printf("1 - pop\n");
   printf("2 - top\n");
   printf("3 - is empty\n");
   printf("4 - exit\n");
}
```

6 Указатели и массивы

6.1 Указатели

Указатель — это переменная, которая содержит некоторый адрес памяти.

Объявление указателя:

```
int a = 3;
int* pa = &a;
```

Оператор & — взятие адреса переменной.

Оператор * (разыменование указателя) — обращение по адресу, который хранится в указателе.

Листинг 22: Демонстрация использования указателей

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

int *a;

int main()
{
  int *b;
  printf("%p\n", a);
  printf("%p\n", b);
  int x = 10;
  int *c = &x;
  printf("%d\n", *c);
  x++;
  printf("%d\n", *c);
  *c = 12;
  printf("%d\n", x);
  return 0;
}
```

Вывод

```
(nil)
```

```
0x7813d20c9af0
10
11
12
```

Если указатель локальный и не проинициализирован, то в нем содержится «мусор». Однако неинициализированный указатель, объявленный глобально, хранится в `bss` и потому его значение равно нулю по умолчанию.

Проинициализировать указатель можно макросом `NULL` (это просто число 0):

```
int* pa = NULL;
```

Разыменование такого указателя приведет к ошибке времени выполнения
`Segmentation fault (core dumped)`

6.2 Константные указатели и указатели на константу

Рассмотрим фрагмент кода:

```
int a = 2, b = 3;
int* p;
p = &a;
p = &b;
*p = 7;
```

После выполнения этих инструкций в переменной `a` будет 2, в переменной `b` — 7.

Добавим слово `const` перед объявлением указателя:

```
1 int a = 2, b = 3;
2 int const* p;
3 p = &a;
4 p = &b;
5 *p = 7;
```

Теперь `p` — это указатель на константу. В результате попытки скомпилировать этот код получим ошибку компиляции в строке 5, потому что попытка изменить константу нелегальна.

Поместим теперь слово `const` чуть правее:

```
1 int a = 2, b = 3;
2 int* const p;
3 p = &a;
4 p = &b;
5 *p = 7;
```

Теперь у нас `p` — это константный указатель. Теперь получаем ошибку компиляции в строках 3 и 4, потому что нельзя менять сам указатель. А менять память, на которую он указывает, допустимо, поэтому в 5 строке ошибки компиляции не будет.

6.3 Связь указателей с массивами

Отметим, что объявление указателя и динамического массива на куче совпадают друг с другом. На самом деле в объявлении динамического массива

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

переменная `a` является указателем и содержит в себе адрес начала массива. Таким образом, разыменование `*a` даст нам нулевой элемент массива. Далее, операция `a+1` преобразуется компилятором в `a+sizeof(тип)`, таким образом, `a+1` является указателем на первый элемент массива, а `*(a+1)` — сам первый элемент массива.

Листинг 23: Двойственность массивов и указателей

```
1 #include <stdio.h>
2 #define SIZE 10
3
4 int main()
5 {
6   int arr[SIZE];
7   for (int i = 0; i < SIZE; i++)
```

```
 8    {
 9      arr[i] = i;
10    }
11    int *p = arr;
12    printf("%d\n", arr[4]);
13    printf("%d\n", p[4]);
14    printf("%d\n", *(p + 4));
15    printf("%d\n", *(arr + 4));
16    printf("%p\n", p + 4);
17    printf("%p\n", arr + 4);
18    printf("%d\n", ++*p);
19    printf("%d\n", ++*arr);
20    printf("%d\n", *++p);
21    printf("%ld\n", p-arr);
22    //printf("%d\n", *++arr);
23    return 0;
24  }
```

Вывод

```
4
4
4
4
0x7ffdafc0e280
0x7ffdafc0e280
1
2
1
1
```

Пояснения

1. Строки 12, 13 демонстрируют, что и к имени массива, и к имени указателя, можно применять оператор [ ], и это приводит к идентичным результатом.

2. Того же эффекта можно достичь, прибавив к имени массива или указателя целое число, затем разыменовав полученный указатель (строки 14, 15).

3. Строки 16 и 17 показывают, что выражения `p+4` и `arr+4` приводят к одному и тому же адресу.

4. В строке 18 мы берем указатель на нулевой элемент массива, разыменовываем его и увеличиваем на 1. Т.к. элемент `p[0]` был равен нулю, то в строке 18 выведется число 1.

5. В строке 19 мы вновь берем указатель на нулевой элемент массива и делаем все то же, что было в строке 18. Но т.к. в 18 строке мы уже увеличили `p[0]` на единицу, то теперь получим ответ 2.

6. В строке 20 мы сперва увеличиваем `p` на 1 (делая по сути `p=p+1`, а потом разыменовываем получившийся указатель. Поэтому выводится значение `arr[1]==p[0]`.

7. В 21 строке мы находим разность между `p` и `arr`. Т.к. `p` теперь равно `arr+1`, то понятно, что разность равна 1.

8. Закомментированная строка 22 показывает, что, несмотря на схожесть массивов и указателей, это не одно и то же: имя массива намертво завязано на его нулевом элементе. Попытка изменить его приводит к ошибке компиляции.

6.4 Аргументы командной строки

Функция `main` получает от операционной системы аргументы командной строки, т.е. те строки, которые пишутся в терминале после имени программы при ее запуске. Функция `main` получает число `argc` — количество аргументов и `char* argv[]` — массив указателей на `char`, т.е. массив строк. Элемент `argv[0]` — это, как правило, имя исполняемого файла. Последующие элементы массива `argv[1],`

`argv[2], ...` — это сами аргументы командной строки. Размер `argv` — это и есть число `argc`.

Программа 24 выводит все аргументы, которые переданы программе в командной строке.

Листинг 24: Вывод аргументов командной строки

```
#include <stdio.h>

int main(int argc, char* argv[])
{
        for(int i=1; i<argc; i++)
        {
                printf("%s\n", argv[i]);
        }
        return 0;
}
```

6.5 Указатель на массив и массив указателей

Материал предыдущего пункта провоцирует нашу фантазию на смелый эксперимент: помимо массива указателей `char* argv[]`[3], из звездочек и скобочек можно еще соорудить конструкцию вида `int (*a)[]`. Проникшись дуализмом указателей и массивов, мы можем придти к мысли, что эти два типа полностью эквиваленты: `char* argv[] = char (*argv)[]`, и оба выражают идею «массива массивов».

Однако попытка присвоить одно значение другому наталкивается на стену непонимания со стороны компилятора. Следовательно, это разные типы.

Прояснить это вопрос поможет программа из листинга 25.

Листинг 25: Как хранятся в памяти указатели на массивы и массивы указателей

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

[3]В силу приоритета операторов `char* argv[] = char* (argv[])`

```c
#include <stdlib.h>

int main()
{
  char *arr[3];
  arr[0] = calloc(10, sizeof(char));
  arr[1] = calloc(10, sizeof(char));
  arr[2] = calloc(10, sizeof(char));
  for (int i = 0; i < 10; i++)
    arr[1][i] = i;
  for (int i = 0; i < 10; i++)
    arr[1][i] = i * 10;
  for (int i = 0; i < 10; i++)
    arr[1][i] = i * 100;
  char **array_of_pointers = calloc(3, sizeof(char *));
  array_of_pointers[0] = arr[0];
  array_of_pointers[1] = arr[1];
  array_of_pointers[2] = arr[2];
  for (int i = 0; i < 3; i++)
  {
    printf("array_of_pointers[%d] = %p\n",
    i, *(array_of_pointers + i));
    for (int j = 0; j < 3; j++)
      printf("array_of_pointers[%d][%d]=%d\t"
        "&array_of_pointers[%d][%d]=%p\n",
        i, j, array_of_pointers[i][j],
        i, j, &array_of_pointers[i][j]);
  }
  int arr1[2][5] = {{1, 2, 3, 4, 5}, {10, 20, 30, 40, 50}};
  int(*ptr_to_array)[5];
  ptr_to_array = &arr1[0];
  for (int i = 0; i < 2; i++)
  {
    printf("Array number = %d\t pointer to array=%p\n",
    i, ptr_to_array);
```

```
    for (int j = 0; j < 5; j++)
    {
      printf("Element number = %d\t address = %p\t",
      j, (*ptr_to_array + j));
      printf("Value = %d\n", *(*ptr_to_array + j));
    }
    ptr_to_array++;
  }
  free(arr[0]);
  free(arr[1]);
  free(arr[2]);
}
```

Вывод

```
array_of_pointers[0] = 0x62cebc5c42a0
array_of_pointers[0][0]=0       &array_of_pointers[0][0]=0x62cebc5c42a0
array_of_pointers[0][1]=0       &array_of_pointers[0][1]=0x62cebc5c42a1
array_of_pointers[0][2]=0       &array_of_pointers[0][2]=0x62cebc5c42a2
array_of_pointers[1] = 0x62cebc5c42c0
array_of_pointers[1][0]=0       &array_of_pointers[1][0]=0x62cebc5c42c0
array_of_pointers[1][1]=100     &array_of_pointers[1][1]=0x62cebc5c42c1
array_of_pointers[1][2]=-56     &array_of_pointers[1][2]=0x62cebc5c42c2
array_of_pointers[2] = 0x62cebc5c42e0
array_of_pointers[2][0]=0       &array_of_pointers[2][0]=0x62cebc5c42e0
array_of_pointers[2][1]=0       &array_of_pointers[2][1]=0x62cebc5c42e1
array_of_pointers[2][2]=0       &array_of_pointers[2][2]=0x62cebc5c42e2
Array number = 0         pointer to array=0x7ffc2f9a7080
Element number = 0       address = 0x7ffc2f9a7080       Value = 1
Element number = 1       address = 0x7ffc2f9a7084       Value = 2
Element number = 2       address = 0x7ffc2f9a7088       Value = 3
Element number = 3       address = 0x7ffc2f9a708c       Value = 4
Element number = 4       address = 0x7ffc2f9a7090       Value = 5
Array number = 1         pointer to array=0x7ffc2f9a7094
Element number = 0       address = 0x7ffc2f9a7094       Value = 10
Element number = 1       address = 0x7ffc2f9a7098       Value = 20
Element number = 2       address = 0x7ffc2f9a709c       Value = 30
Element number = 3       address = 0x7ffc2f9a70a0       Value = 40
Element number = 4       address = 0x7ffc2f9a70a4       Value = 50
```

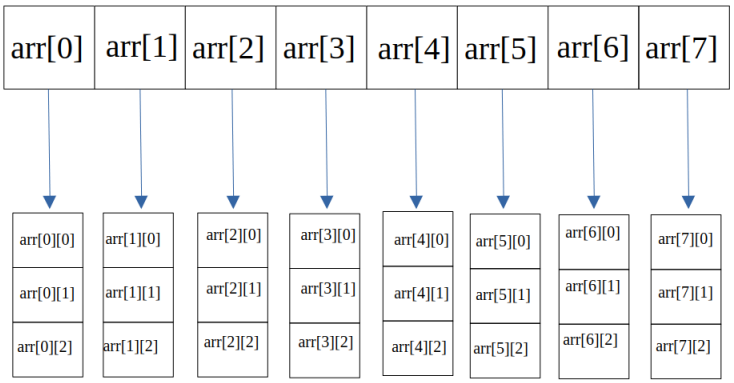


Рис. 5: Массив указателей

Тщательный анализ кода и результатов работы программы позволяют сделать следующие выводы :

1. Каждый элемент массива `arr` представляет собой динамический массив, внутри которого все элементы расположены сплошным образом, один за другим[4]. Адреса элементов этих массивов отличаются друг от друга на один байт — размер `char`. В то же время расстояние между последним элементом одного массива и нулевым элементов следующего за ним массива не равно одному байту.

2. Указатель на указатель и массив указателей суть одно и то же.

3. Все элементы указателя на массив `ptr_to_array` расположены друг за другом, их адреса отличаются на размер базовой переменной, в данном случае `int`.

4. Указатель на массив и статический двумерный массив (статический массив статических массивов) это одно и то же.

Разница в расположении в памяти указателя на массив и массив указателей показана на рис. 6 и 5.

[4]Это общее свойство функций `malloc, calloc, realloc`.

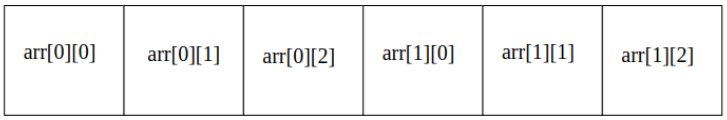


Рис. 6: Указатель на массив

Таким образом, указатель на массив — это статический двумерный массив. Массив указателей — это динамический двумерный массив.

7 Функции

7.1 Создание и использование функций

Функция — это именованный фрагмент кода, который может принимать и возвращать значения.

Объявление функции

`return_type function_name(t1 val1, t2 val 2, ...tn valn),`

где

`return_type` — возвращаемое значение;

`function_name` — имя функции;

`t1, t2, ...tn` — типы 1-го, 2-го ... n-го аргумента функции;

`val1, val2, ..., valn` — имена 1-го, 2-го ... n-го аргумента функции.

Имя функции, тип, количество и порядок аргументов представляют собой `сигнатуру функции`.

Определение функции (т.е. ее «тело», конкретные выполняемые инструкции) может быть совмещено с объявлением, а может находиться отдельно.

В случае, если определение функции и объявления функции не объединены, то имена аргументов функции в ее объявлении можно опустить.

Необходимо объявить функцию до ее первого использования. Определение же функции может быть и ниже по коду, чем ее первое использование.

Возвращение значения из функции выполняется с помощью ключевого слова `return`. После выполнения инструкции `return` функция завершается и никакие другие операторы больше не выполняются.

Листинг 26: Объявление и определение функции объединены

```
#include <stdio.h>

int add(int a, int b)
{
  return a + b;
```

```
}

int main()
{
  printf("Sum of 2 and 3 is %d\n", add(2, 3));
  return 0;
}
```

Листинг 27: Объявление и определение функции разъединены

```
#include <stdio.h>

int add(int, int);

int main()
{
  printf("Sum of 2 and 3 is %d\n", add(2, 3));
  return 0;
}

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

7.2 Передача аргументов по указателю

Напишем функцию `swap`, которая меняет местами значения двух целочисленных аргументов. Первая попытка создать такую функцию представлена в листинге 28. Легко убедиться в том, что такая попытка не приводит нас к успеху: значения переменных так и не поменялись после вызова функции. Причина заключается в том, что функция `swap` работает с копиями передаваемых ей переменных. Аргументы функции и ее локальные переменные хранятся на стеке и удаляются после того, как функция завершила свою работу. Следо-

вательно, все изменения, которые совершает функция над переданными ей аргументами, совершаются над копиями, которые удаляются сразу после завершения работы функции, а сами аргументы остаются неизменными.

Листинг 28: Неудачная попытка замены переменных

```
#include <stdio.h>

void swap(int a, int b)
{
  int tmp = a;
  a = b;
  b = tmp;
}

int main()
{
  int a = 2, b = 3;
  swap(a, b);
  printf("a = %d, b = %d\n", a, b); // a = 2, b = 3
  return 0;
}
```

Решение проблемы: передавать в функцию указатели на переменные. Тогда в функцию будут скопированы переданные указатели, при разыменовании которых мы сможем изменить исходные переменные! Тогда в качестве параметров при вызове фунции мы будем передавать уже адреса переменных. Эта идея реализована в листинге 29.

Листинг 29: Удачная попытка замены переменных

```
#include <stdio.h>

int swap(int* a, int* b)
{
  int tmp = *a;
```

```
    *a = *b;
    *b = tmp;
}

int main()
{
    int a = 2, b = 3;
    swap(&a, &b);
    printf("a = %d, b = %d\n", a, b); // a = 3, b = 2
    return 0;
}
```

Теперь, в частности, понятно, почему в функции `scanf` нужно писать амперсанд перед именем переменной: на самом деле в эту функцию передается указатель на переменную, чтобы функция `scanf` смогла записать изменения в передаваемый аргумент.

7.3 Указатель на функцию

Указатель на функцию содержит адрес, по которому расположены команды функции. По аналогии с массивом, имя функции — это указатель на начало функции.

Листинг 30: Передача функций в качестве аргументов

```
#include <stdio.h>

double f(double x)
{
    return x * x;
}

double g(double x)
{
    return x;
}
```

```
double Integral(double a, double b, int n,
  double (*f)(double x))
{
  double h = (b - a) / n;
  double sum = 0.5*(f(a) + f(b));
  for(int i=1; i<n; i++)
  {
    double x = a+i*h;
    sum += f(x);
  }
  sum *= h;
  return sum;
}

int main()
{
  printf("%lf\n", Integral(0, 3, 1000, f));
  printf("%lf\n", Integral(0, 3, 1000, g));
  return 0;
}
```

7.4 Функции с переменным числом аргументов

Напишем функцию, которая находит максимальное число из переданных ей аргументов, причем сделаем так, чтобы количество аргументов могло бы быть произвольным (2, 3 и далее).

Листинг 31: Функции с переменным количеством аргументов

```
// C program for the above approach
#include <stdarg.h>
#include <stdio.h>

// Variadic function to find the largest number
int LargestNumber(int n, ...)
{
```

```
    // Declaring pointer to the
    // argument list
    va_list ptr;

    // Initializing argument to the
    // list pointer
    va_start(ptr, n);

    int max = va_arg(ptr, int);

    for (int i = 0; i < n-1; i++) {

        // Accessing current variable
        // and pointing to next
        int temp = va_arg(ptr, int);
        max = temp > max ? temp : max;
    }

    // End of argument list traversal
    va_end(ptr);

    return max;
}

// Driver Code
int main()
{
    printf("\n\n Variadic functions: \n");

    // Variable number of arguments
    printf("\n %d ",
    LargestNumber(2, 1, 2));

    printf("\n %d ",
    LargestNumber(3, 3, 4, 5));
```

```
  printf("\n %d ",
  LargestNumber(4, 6, 7, 8, 9));

  printf("\n");

  return 0;
}
```

7.5 Стек вызовов

Пусть у нас есть две функции, `foo()` и `bar()`. Пусть из `main` вызывается `foo()`, а из `foo()` вызывается `bar`. Тогда стек можно схематически представить в виде 7.

Рассмотрим программу из листинга 32. При вызове функции `f` стек примет вид, представленный на рис. 8.

`Frame pointer` указывает на начало стека функции, которая в данный момент выполняется.

`Stack pointer` указывает на конец стека текущей функции.

2, 1 в учейках 3 и 4 сверху — это принимаемые значения (копии).

`retval` — возвращаемое значение.

`retaddress` — адрес возврата (куда идти, когда функция завершит свою работу).

`registers` — содержимое регистров процессора вызывающей функции.

Листинг 32: Демонстрация стека

```
#include <stdio.h>

int f(int x, int y)
{
  int d = x + y;
  int s = x - y;
  return d + s;
}
```

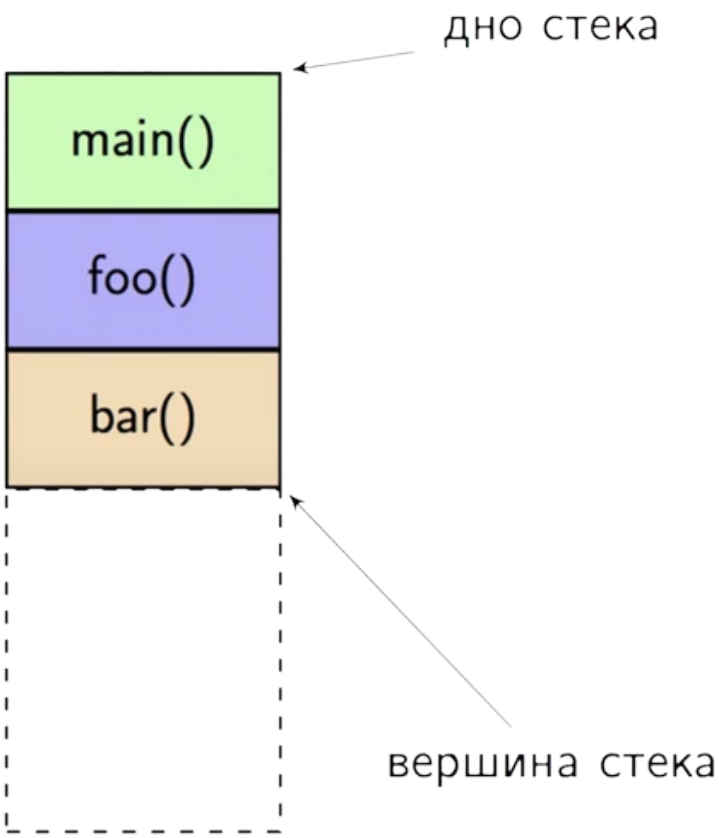


Рис. 7: Стек вызовов

```
int main()
{
  int a = 1;
  int b = 2;
  int z = f(a, b);
  return 0;
}
```

Листинг 33: Переполнение буфера

```
#include <stdio.h>

int MaliciousCode(int x, int y)
```

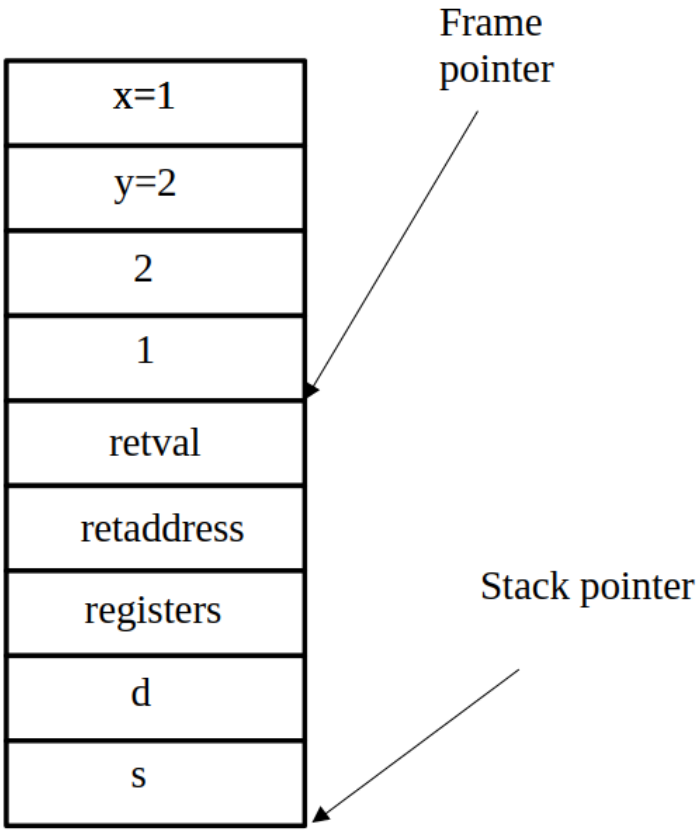


Рис. 8: Структура стека

```
{
  printf("Hey!\n");
  return 0;
}

int GoodCode()
{
  int* m[1];
  m[3] = (int*)MaliciousCode;
  return 0;
}

int main()
{
  GoodCode();
  return 0;
}
```

7.6 Передача массивов в функции, возврат массива из функции

Задача 1. Написать функцию `CreateArray`, которая создает массив длины n и записывает туда случайные числа. Написать функцию `PrintArray`, которая выводит элементы массива на экран.

Решение задачи показано в листинге

Листинг 34: Передача массива в функцию и возвращение массива из функции

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int *CreateArray(int n)
5 {
6   int *a = calloc(n, sizeof(int));
```

```
 7    for (int i = 0; i < n; i++)
 8      a[i] = rand() % 10;
 9    return a;
10  }
11
12  void PrintArray(const int* a, int n)
13  {
14    for(int i=0; i<n; i++)
15      printf("%d ", a[i]);
16    printf("\n");
17  }
18
19  int main()
20  {
21    int n;
22    scanf("%d", &n);
23    int *arr = CreateArray(n);
24    PrintArray(arr, n);
25    free(arr);
26    return 0;
27  }
```

Обратите внимание: в 12 строке передается указатель на константу, чтобы компилятор выдал ошибку в случае, если в коде случайно окажется инструкция вида

```
a[i] = 777;
```

Задача 2. Написать программу умножения матрицы на вектор.

Листинг 35: Умножение матрицы на вектор (статический двумерный массив)

```
#include <stdio.h>
#include <stdlib.h>
#define ROWS 5
#define COLS 4
```

```
int CreateVector(int a[])
{
  for (int i = 0; i < COLS; i++)
    a[i] = rand() % 10;
}

void PrintVector(const int *a, int n)
{
  printf("Vector: \n");
  for (int i = 0; i < n; i++)
    printf("%d ", a[i]);
  printf("\n");
}

void CreateMatrix(int (*mat)[COLS])
{
  for (int i = 0; i < ROWS; i++)
    for (int j = 0; j < COLS; j++)
      mat[i][j] = rand() % 10;
}

void PrintMatrix(int (*mat)[COLS])
{
  printf("Matrix: \n");
  for (int i = 0; i < ROWS; i++)
  {
    for (int j = 0; j < COLS; j++)
      printf("%d ", mat[i][j]);
    printf("\n");
  }
}

void MatVecMult(int (*mat)[COLS], int *vec, int *res)
{
```

```
  for (int i = 0; i < ROWS; i++)
  {
    res[i] = 0;
    for (int j = 0; j < COLS; j++)
      res[i] += mat[i][j] * vec[j];
  }
}

int main()
{
  int vec[COLS];
  int mat[ROWS][COLS];
  int res[ROWS];
  CreateVector(vec);
  PrintVector(vec, COLS);
  CreateMatrix(mat);
  PrintMatrix(mat);
  MatVecMult(mat, vec, res);
  PrintVector(res, ROWS);
  return 0;
}
```

Обратите внимание, что в силу выводов стр. 51 в качестве двумерного статического массива в функцию можно передавать `int mat[][COLS]` наравне с `int (*mat)[COLS]`.

Необходимо избавиться от искушения писать код навроде такого:

```
int* CreateVector()
{
  int a[COLS];
  for (int i = 0; i < COLS; i++)
  a[i] = rand() % 10;
  return a;
}
```

В этом случае возвращается локальная переменная `a`, которая находится на стеке функции `CreateVector`, а потому будет уничто-

жена, как только функция прекратит свою работу.

```
#include <stdio.h>
#include <stdlib.h>
#define ROWS 5
#define COLS 4

int *CreateVector(int n)
{
  int *a = calloc(n, sizeof(int));
  for (int i = 0; i < COLS; i++)
    a[i] = rand() % 10;
  return a;
}

void PrintVector(const int *a, int n)
{
  printf("Vector: \n");
  for (int i = 0; i < n; i++)
    printf("%d ", a[i]);
  printf("\n");
}

int **CreateMatrix(int r, int c)
{
  int **mat = calloc(r, sizeof(int *));
  for (int i = 0; i < r; i++)
  {
    mat[i] = calloc(c, sizeof(int));
    for (int j = 0; j < c; j++)
    mat[i][j] = rand() % 10;
  }
  return mat;
}
```

```
void PrintMatrix(int **mat, int r, int c)
{
  printf("Matrix: \n");
  for (int i = 0; i < r; i++)
  {
    for (int j = 0; j < c; j++)
    printf("%d ", mat[i][j]);
    printf("\n");
  }
}

int *MatVecMult(int **mat, int *vec, int r, int c)
{
  int *res = calloc(r, sizeof(int));
  for (int i = 0; i < r; i++)
  {
    res[i] = 0;
    for (int j = 0; j < c; j++)
    res[i] += mat[i][j] * vec[j];
  }
  return res;
}

int** DeleteMatrix(int*** mat, int r)
{
  for(int i=0; i<r; i++)
  {
    free((*mat)[i]);
  }
  free(*mat);
  return NULL;
}

int* DeleteVector(int** vec)
{
```

```
    free(*vec);
    return NULL;
}

int main()
{
    int *vec = CreateVector(COLS);
    PrintVector(vec, COLS);
    int **mat = CreateMatrix(ROWS, COLS);
    PrintMatrix(mat, ROWS, COLS);
    int *res = MatVecMult(mat, vec, ROWS, COLS);
    PrintVector(res, ROWS);
    mat = DeleteMatrix(&mat, ROWS);
    vec = DeleteVector(&vec);
    res = DeleteVector(&res);

    return 0;
}
```

7.7 Сложные объявления

Из звездочек, квадратных скобок, круглых скобок и модификаторов `const` можно составить много сложных объявлений.

Используйте сайт для того, чтобы переводить объявление с языка Си на человеческий язык, и наоборот.

Также можно использовать правило «направо-налево»: найти имя, идти вправо до конца, либо до первых круглых скобок, затем идти влево до конца либо до первых круглых скобок, затем опять направо до конца или до первых круглых скобок (игнорируя те круглые скобки, о которые мы уже споткнулись), и далее, пока строка не кончится.

Пример:

```
char * const (*(* const bar)[5])(int )
```

Расшифровка: `bar` — это постоянный указатель на массив из 5 указателей на функцию, принимающую целое число, и возвращающую постоянный указатель на символ.

8 Строки

8.1 Константные указатели и указатели на константу

Рассмотрим фрагмент кода:

```
int a = 2, b = 3;
int* p;
p = &a;
p = &b;
*p = 7;
```

После выполнения этих инструкций в переменной `a` будет 2, в переменной `b` — 7.

Добавим слово `const` перед объявлением указателя:

```
1 int a = 2, b = 3;
2 int const* p;
3 p = &a;
4 p = &b;
5 *p = 7;
```

Теперь `p` — это указатель на константу. В результате попытки скомпилировать этот код получим ошибку компиляции в строке 5, потому что попытка изменить константу нелегальна.

Поместим теперь слово `const` чуть правее:

```
1 int a = 2, b = 3;
2 int* const p;
3 p = &a;
4 p = &b;
5 *p = 7;
```

Теперь у нас `p` — это константный указатель. Теперь получаем ошибку компиляции в строках 3 и 4, потому что нельзя менять сам указатель. А менять память, на которую он указывает, допустимо, поэтому в 5 строке ошибки компиляции не будет.

8.2 Что такое строка

Строка — это массив символов (элементов типа `char`), причем последний символ должен иметь код ноль (число 0, или '\0').

Листинг 36: Объявление строк

```
1  const char *s1 = "Hello, world!";
2  char s2[100] = "Hello, world!";
3  char *s3 = malloc(100);
4  s1 = s2; // OK
5  s2 = s1; // CE
6  s3 = s1; // memory leak
7  s1[0] = 'h'; //CE
8  s2[0] = 'h'; //OK
9  printf("%d\n", sizeof(s1)); // 8
10 printf("%d\n", sizeof(s2)); // 100
11 free(s3);
```

Комментарии к листингу 36:

1. В строке 1 объявляется указатель на `char`, который инициализируется строковым литералом. Этот строковый литерал помещается в ту часть инициализированного сегмента данных, которая допускает только чтение (см. стр. 34). Именно поэтому присваивание в строке 7 обречено на провал: это будет ошибка компиляции, если в строке 1 `s1` объявлен как указатель на константу, и ошибка времени выполнения, если модификатор `const` опущен.

2. В строке 4 мы пользуемся дуализмом между указателями и массивами, что позволяет нам присвоить указателю имя массива.

3. В строке 5 мы пытаемся сделать обратное присваивание, которое приводит к ошибке компиляции. Причина заключается в том, что имя массива «намертво» связано с областью памяти, на которую оно указывает.

Название функции	Что она делает
`strcpy(dest, source)`	Копирует `source` в tt dest
`strcat(dest, source)`	Дописывает `source` в конец `dest`
`strlen(s)`	Возвращает длину `s` (без '\0')
`strcmp(s1, s2)`	Сравнивает `s1` и `s2`, возвращает 0, 1, -1
`strchr(str, ch)`	Ищет символ в строке
`strstr(str, substr)`	Ищет подстроку в строке
`strspn(s1, s2)`	Ищет макс. префикс `s1`, состоящий только из символов `s2`
`strcspn(s1, s2)`	То же, только не из символов `s2`
`strpbrk(s1, s2)`	Ищет первое вхождение в `s1` любого из символов `s2`

Таблица 2: Функции для работы со строками

4. Присваивание в строке 6 приводит к утечке памяти, потому что теряется доступ к области памяти, выделенной с помощью функции `malloc` в строке 3. Кроме того, `free` в строке 11 тогда будет освобождать read-only дата сегмент, что является плохой идеей и также приведет к ошибке времени выполнения.

5. Строки 9 и 10 иллюстрируют еще одно отличие между массивом и указателем: размер указателя равен 8 байтам для 64-битной системы, а размер массива равен количеству байтов во всем массиве.

8.3 Функции для работы со строками

Основные функции для работы со строками показаны в таблице 2.

Домашнее задание. Посмотреть на на сайте синтаксис и примеры функций из таблицы 2.

Листинг 37: Реализации некоторых функций работы со строками

```
int str_len(char const* str)
{
```

```
  int len = 0;
  while (str[len++]);
  return len-1;
}

void str_copy(char* dest, char const* source)
{
  while (*dest++ = *source++);
}

void str_concat(char* dest, char const* source)
{
  while (*dest++);
  str_copy(dest-1, source);
}

int str_compare(char const* str1, char const* str2)
{
  while (*str1 && *str2)
  {
    if (*str1 > *str2)
      return 1;
    if (*str1 < *str2)
      return -1;
    str1++;
    str2++;
  }
  if (*str1)
    return 1;
  if (*str2)
    return -1;
  return 0;
}
```

9 Препроцессор

Директива препроцессора — это команда для препроцессора. Директивы начинаются со знака диеза #. Точка с запятой после директивы не ставится, так как предполагается, что директива занимает одну строку и окончание строки есть конец директивы. В случае необходимости можно перенести на следующую строку, поставив знак \.

9.1 Макросы

Макросом в языке Си называется всякий фрагмент кода, имеющий имя. Препроцессор в ходе обработки текста программы заменяет имя макроса на соответствующий ему фрагмент кода.

С помощью макросов можно задавать константы:

```
#define COMST_PI 3.14159
```

Также можно создавать макросы с параметрами, некоторые особенности работы с которыми иллюстрирует листинг 38.

Листинг 38: Макрос с параметрами

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  #define MAX(a, b) ((a) > (b)) ? (a) : (b)
5  #define MAX1(a, b) a > b ? a : b
6
7  int main()
8  {
9     printf("%d\n", MAX(2, 3));
10    printf("%d\n", MAX(2, 1 < 0));
11    printf("%d\n", MAX1(2, 1 < 0));
12    int a = 2, b = 3;
13    printf("%d\n", MAX(a++, b++));
14    printf("%d %d\n", a, b);
15    return 0;
```

```
16 }
```

Вывод.

```
3
2
0
4
3 5
```

Пояснения.

1. Сопоставление выводов программы в строках 10 и 11 обосновывает необходимость ставить скобки в макросах.

2. Необходимо помнить, что макросы для препроцессора — это команды «скопировать и вставить», что иллюстрируют строки 13 и 14 программы. Вызов макроса в 13 строке приводит к генерации препроцессором следующего кода:
   `((a++)>(b++)) ? (a++) : (b++);`
   При выполнении кода сравнивается 2 и 3. Так как условие не выполняется, то далее производится инкремент `b++`, `b` становится равным 4, и это значение возвращается в 13 строке. Далее инкрементируются `a` и `b`. В результате `a` будет равно 3, а `b` примет значение 5.

9.2 Условная компиляция

Директивы для условной компиляции

1. `#ifdef macro_name`. Выполнять следующий код, если имя `macro_name` было ранее определено директивой `#define`.

2. `#ifndef macro_name`. Не выполнять следующий код, если имя `macro_name` было ранее определено директивой `#define`.

3. Конструкция
   ```
   #if cond
   Выполнить, если cond истина
   ```

```
#elif another_cond
Выполнить, если another_cond истина
#else
Выполнить, если ни одно из условий не подходит
```

Вне зависимости от того, как начинался блок условной компиляции, заканчиваться он должен директивой `#endif`.

Есть также директива `#undef`, которая позволяет отменить эффект директивы #define.

Сферы применения условной компиляции:

1. Временно закомментировать часть кода.
2. Организовать отладочный вывод, который будет производится только в том случае, если определен макрос `DEBUG`.
3. Компиляция платформенно-зависимого кода.
4. Распределение исходного кода программы по нескольким файлам.

Проиллюстрируем последние два сценария.

В листинге 39 продемонстрирована работа платформенно-зависимой функции `sleep`, используемой для задержки работы программы на какое-то время. В ОС семейства `Windows` данная функция задекларирована в заголовочном файле `windows.h` и время задержки задается в миллисекундах. В `UNIX`-подобных системах заголовочный файл называется `unistd.h` и время задается в секундах.

Листинг 39: Компиляция платформенно-зависимого кода (усыпление программы)

```
#include <stdio.h>
#ifdef __linux__
#include <unistd.h>
#elif _WIN32
#include <windows.h>
#endif
```

```
int main()
{
  printf("The program will sleep for 5 sec\n");
#ifdef __linux__
  sleep(5);
#elif _WIN32
  sleep(5000);
#endif
  return 0;
}
```

Рассмотрим распределение программы по нескольким исходным файлам:

Листинг 40: Файл main.c

```
#include <stdio.h>

#include"f.h"
#include"g.h"

int main()
{

  return 0;
}
```

Листинг 41: Файл f.h

```
#ifndef F_H
#define F_H
#include"g.h"

void f();

#endif
```

Листинг 42: Файл g.h

```
#ifndef G_H
#define G_H

const double pi = 3.14159;

void g();

#endif
```

Файлы с реализациями функций `f.c, g.c` опущены для краткости.

Компиляция программы

```
gcc main.c f.c g.c -o main
```

Если убрать директивы препроцессора из файлов `f.h, g.h`, то будет переопределение константы `pi` из-за двойного включения в `main.c`.

Отметим, что в директиве `#include` имя файла может указываться в треугольных кавычках или в обычных. В первом случае файл ищется сначала в специальном каталоге с заголовочными файлами и, в случае неудачи, в папке с исходным кодом. Во втором случае файл изначально ищется в папке с исходным кодом, и если его там нет, то поиск осуществляется по глобальному каталогу, в котором хранятся заголовочные файлы библиотечных функций.

Директивы `#pragma` зависят от компилятора. Код с такими директивами не является портабельным в строгом смысле слова.

`#pragma once` — замена конструкции `#ifndef ...` для защиты от двойного включения.

9.3 Замеры времени работы программы

Применим наши знания директив препроцессора, чтобы написать макрос для замера времени работы программы.

Листинг 43: Макрос для замера времени

```
#ifndef _TIMER_H_
```

```
#define _TIMER_H_

#include <sys/time.h>

#define GET_TIME(now) { \
        struct timeval t; \
        gettimeofday(&t, NULL); \
        now = t.tv_sec + t.tv_usec/1000000.0; \
}

#endif
```

Два типа времени: `wall time` (физическое время «настенных» часов) и `cpu time` (количество тактов процессора). Макрос в листинге 43 замеряет wall time.

Отличие wall и cpu time показаны в листинге 44. Первый вывод дает нам очень маленькое число, так как cpu time считает такты процессора, а когда программа спит 5 секунд, то процесс в это время ничего не делает.

Листинг 44: Отличие wall time и cpu time

```
#include <stdio.h>
#include <unistd.h>
#include <time.h>
#include "timer.h"

int main()
{
        clock_t start = clock();
        double begin, finish;
        GET_TIME(begin);
        sleep(5);
        GET_TIME(finish);
        clock_t end = clock();
        double cpu_time_used =
                ((double)(end - start)) / CLOCKS_PER_SEC;
```

```
        double wall_time_used = finish - begin;
        printf("%lf\n", cpu_time_used); //0.0007
        printf("%lf\n", wall_time_used); //5.0002
        return 0;
}
```

10 Структуры, объединения, перечисления

10.1 Ключевое слово `typedef`

Ключевое слово `typedef` позволяет определить новый тип данных.

```
typedef existing_name alias_name;
```

Примеры:

```
typedef unsigned int uint;
typedef char * const (*(* const confusing_type)[5])(int );
uint x;
confusing_type y;
```

10.2 Перечисления

Перечисления позволяют компактно объявить несколько целочисленных констант:

```
enum week{Mon, Tue, Wed, Thu = 4, Fri, Sat, Sun};
printf("%d\n", Mon); //0
printf("%d\n", Tue); //1
printf("%d\n", Wed); //2
printf("%d\n", Thu); //4
printf("%d\n", Fri); //5
printf("%d\n", Sat); //6
printf("%d\n", Sun); //7
```

10.3 Структуры

Декларация структуры

```
struct structure_name {
        data_type member_name1;
        data_type member_name1;
        ...
```

```
        ...
};
```

Объявление переменных

```
struct structure_name variable1, variable2, .......;
```

Доступ к полям структуры:

```
structure_name.member1;
strcuture_name.member2;
```

Объявление указателей на структуру:

```
        struct structure_name *variable1, *variable2, .......;
```

Два эквивалентных способа доступа к полям структуры через указатели:

```
        (*structure_name).member1;
        strcuture_name->member2;
```

10.4 Объединения

Работа с объединениями ничем не отчается от работы со структурами. Вместо ключевого слова `struct` используем `union`.

Принципиальное отличие объединений от пересечений состоит в том, что все поля хранятся в одной и той же области памяти. Поэтому одно и то же время в объединении хранится только одно поле.

10.5 Пример

Листинг 45: Файл `point.h`

```
#ifndef POINT_H
#define POINT_H

typedef struct Point
{
        double x, y;
```

```
}POINT;

void PrintPoint(const POINT* p);
int IsEqualPoints(const POINT* p1, const POINT* p2);
double CalculateDistance(const POINT* p1, const POINT* p2);

#endif
```

Листинг 46: Файл `point.c`

```
#include "point.h"
#include <stdio.h>
#include <math.h>

void PrintPoint(const POINT* p)
{
        printf("[%.2lf, %.2lf]\n", p->x, p->y);
}

int IsEqualPoints(const POINT* p1, const POINT* p2)
{
        return p1->x==p2->x && p1->y==p2->y;
}

double CalculateDistance(const POINT* p1, const POINT* p2)
{
        return sqrt(pow(p1->x-p2->x,2)+ pow(p1->y-p2->y,2));
}
```

Листинг 47: Файл `point.h`

```
#pragma once

#include "point.h"

typedef struct Circle {
        POINT center;
```

```
        double radius;
}CIRCLE;

void PrintCircle(const CIRCLE* c);
int IsEqualCircles(const CIRCLE* c1, const CIRCLE* c2);
int IsConcentricCircles(const CIRCLE* c1, const CIRCLE* c2);
int IsNestedCircles(const CIRCLE* c1, const CIRCLE* c2);
```

Листинг 48: Файл `point.c`

```
#include "circle.h"
#include <stdio.h>
#include <math.h>


void PrintCircle(const CIRCLE* c)
{
        printf("center = ");
        PrintPoint(&c->center);
        printf("radius = %.2lf\n", c->radius);
}

int IsEqualCircles(const CIRCLE* c1, const CIRCLE* c2)
{
        return IsEqualPoints(&c1->center, &c2->center) &&
                c1->radius == c2->radius;
}

int IsConcentricCircles(const CIRCLE* c1, const CIRCLE* c2)
{
        return IsEqualPoints(&c1->center, &c2->center);
}

int IsNestedCircles(const CIRCLE* c1, const CIRCLE* c2)
{
        double maxim = fmax(c1->radius, c2->radius);
```

```
        double minim = fmin(c1->radius, c2->radius);
        return maxim>=minim +
                CalculateDistance(&c1->center, &c2->center);
}
```

Листинг 49: Файл `circle_array.h`

```
#pragma once
#include "circle.h"

typedef struct CircleArray {
        CIRCLE* arr;
        int top;
        int max_size;
}CIRCLE_ARRAY;

void InitCircleArray(CIRCLE_ARRAY* arr, int size);
void AddCircleToArray(CIRCLE_ARRAY* arr,
        const CIRCLE* value);
void DeleteCircleFromArray(CIRCLE_ARRAY* arr,
        const CIRCLE* value);
void PrintCircleArray(const CIRCLE_ARRAY* arr);
void FindConcetricCircles(const CIRCLE_ARRAY* arr);
void FindNestedCircles(const CIRCLE_ARRAY* arr);
```

Листинг 50: Файл `circle_array.c`

```
#include "circle_array.h"
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>

void InitCircleArray(CIRCLE_ARRAY* arr, int size)
{
  arr->arr =
    (CIRCLE*)calloc(arr->max_size = size, sizeof(CIRCLE));
  arr->top = 0;
```

```
}

void AddCircleToArray(CIRCLE_ARRAY* arr, CIRCLE* value)
{
  if (arr->top == arr->max_size)
  {
    arr->max_size = arr->max_size * 2 + 1;
    CIRCLE* temp =
      (CIRCLE*)calloc(arr->max_size, sizeof(CIRCLE));
    for (int i = 0; i < arr->top; ++i)
    {
      temp[i] = arr->arr[i];
    }
    free(arr->arr);
    arr->arr = temp;
  }
  arr->arr[arr->top++] = *value;
}

void DeleteCircleFromArray(CIRCLE_ARRAY* arr,
  const CIRCLE* value)
{
  int i=0;
  for (; i < arr->top; ++i)
  {
    if (IsEqualCircles(&arr->arr[i], value))
    break;
  }
  if (i != arr->top)
  {
    arr->arr[i] = arr->arr[--arr->top];
  }
}

void PrintCircleArray(const CIRCLE_ARRAY* arr)
```

```
{
  for (int i = 0; i < arr->top; ++i)
  {
    printf("Circle %d:\n", i + 1);
    PrintCircle(&arr->arr[i]);
  }
}

void FindConcetricCircles(const CIRCLE_ARRAY* arr)
{
  for (int i = 0; i < arr->top; ++i)
  {
    int counter = 0;
    printf("Concentric for:\n");
    PrintCircle(&arr->arr[i]);
    for (int j = i + 1; j < arr->top; ++j)
    {
      if (IsConcentricCircles(&arr->arr[i], &arr->arr[j]))
      {
        PrintCircle(&arr->arr[j]);
        counter++;
      }
    }
    if (!counter)
    {
      printf("No concentric\n");
    }
    counter = 0;
  }
}

void FindNestedCircles(const CIRCLE_ARRAY* arr)
{
  for (int i = 0; i < arr->top; ++i)
  {
```

```
      int counter = 0;
      printf("Nested for:\n");
      PrintCircle(&arr->arr[i]);
      for (int j = i + 1; j < arr->top; ++j)
      {
        if (IsNestedCircles(&arr->arr[i], &arr->arr[j]))
        {
          PrintCircle(&arr->arr[j]);
          counter++;
        }
      }
      if (!counter)
      {
        printf("No nested\n");
      }
      counter = 0;
    }
  }
```

11 Односвязный список

Листинг 51: list.h

```
#pragma once

typedef struct Node{
  int value;
  struct Node* next;
}NODE;

void Add2List(NODE** pphead, int value);
void PrintList(const NODE* phead);
NODE* DeleteList(NODE* phead);
```

Листинг 52: list.c

```
#include "list.h"
#include <stdio.h>
#include <stdlib.h>

void Add2List(NODE** pphead, int value)
{
  while (*pphead)
  {
    if ((*pphead)->value > value)
    break;
    pphead = &((*pphead)->next);
  }
  NODE* pnew = (NODE*)malloc(sizeof(NODE));
  pnew->value = value;
  pnew->next = *pphead;
  *pphead = pnew;
}

void PrintList(const NODE* phead)
```

```
{
  while (phead)
  {
    printf("%5d", phead->value);
    phead = phead->next;
  }
  printf("\n");
}

NODE* DeleteList(NODE* phead)
{
  if (phead)
  {
    DeleteList(phead->next);
    free(phead);
  }
  return NULL;
}
```

12 Бинарное дерево поиска

Бинарное дерево поиска — структура данных, состоящая из корня, левого поддерева, содержащего элементы меньше, чем корень, и правого поддерева, содержащего элементы, которые больше корневого. При этом левое поддерево и правое поддерево должны быть также бинарными деревьями поиска (рекурсивное определение). Также поддеревья могут быть и пустыми множествами.

Листинг 53: Файл btree.h

```
#pragma once

typedef struct node
{
  int value;
  struct node* left, * right;
}NODE;

NODE* Add2Tree(NODE* root, int value);
NODE* DeleteTree(NODE* root);
void In(const NODE* root);
void Pre(const NODE* root);
void Post(const NODE* root);
void PrintTreeOnSide(const NODE* root, int level);
NODE* DeleteFromTree(NODE* root, int value);
```

Листинг 54: Файл btree.c

```
#include "btree.h"
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>

NODE* Add2Tree(NODE* root, int value)
{
  if (!root)
```

```
  {
    root = (NODE*)malloc(sizeof(NODE));
    root->value = value;
    root->left = root->right = NULL;
  }
  else if (value < root->value)
  {
    root->left = Add2Tree(root->left, value);
  }
  else if (value > root->value)
  {
    root->right = Add2Tree(root->right, value);
  }
  return root;
}

NODE* DeleteTree(NODE* root)
{
  if (root)
  {
    root->left = DeleteTree(root->left);
    root->right = DeleteTree(root->right);
    free(root);
  }
  return NULL;
}

void In(const NODE* root)
{
  if (root)
  {
    In(root->left);
    printf("%5d", root->value);
    In(root->right);
  }
```

```
}

void Pre(const NODE* root)
{
  if (root)
  {
    printf("%5d", root->value);
    Pre(root->left);
    Pre(root->right);
  }
}

void Post(const NODE* root)
{
  if (root)
  {
    Post(root->left);
    Post(root->right);
    printf("%5d", root->value);

  }
}

void PrintTreeOnSide(const NODE* root, int level)
{
  if (root)
  {
    PrintTreeOnSide(root->right, level + 1);
    for (int i = 0; i < level; ++i)
    printf("\t");
    printf("%5d\n", root->value);
    PrintTreeOnSide(root->left, level + 1);
  }
}
```

```
void Replace(NODE** elem, int* value)
{
  if ((*elem)->left)
  Replace(&((*elem)->left), value);
  else
  {
    *value = (*elem)->value;
    NODE* temp = *elem;
    *elem = (*elem)->right;
    free(temp);
  }
}

NODE* DelNode(NODE* root)
{
  if (root->left == NULL && root->right == NULL)
  {
    free(root);
    return NULL;
  }
  if (root->left && !root->right)
  {
    NODE* temp = root;
    root = root->left;
    free(temp);
    return root;
  }
  if (!root->left && root->right)
  {
    NODE* temp = root;
    root = root->right;
    free(temp);
    return root;
  }
  if (root->left && root->right)
```

```
  {
    int value;
    Replace(&root->right, &value);
    root->value = value;
  }
}

NODE* DeleteFromTree(NODE* root, int value)
{
  if (root)
  {
    if (root->value == value)
    {
      root = DelNode(root);
    }
    else if (value < root->value)
    {
      root->left = DeleteFromTree(root->left, value);
    }
    else
    {
      root->right = DeleteFromTree(root->right, value);
    }
  }
  return root;
}
```

13 Файлы

Файл в языке Си представляется указателем на объект типа `FILE`. Указатель инициализируется значением, которое возвращает функция `fopen`. Ее первый аргумент — это имя файла. Второй аргумент — это строка. Первый символ строки должен быть буквой `r` (чтение) или `w` (запись). Второй символ — `t` (текстовый, значение по умолчанию) или `b` (бинарный).

После использования файла его нужно обязательно закрыть, вызвав функцию `fclose(f)`, где `f` — указатель на файл. Если не закрыть файл, то часть информации, которую мы туда записали, может быть потеряна. Это связано с тем, что информация не сразу записывается в файл, а сначала поступает в буфер. Когда в буфере накопится достаточно информации, то она вся записывает в файл. Причина такого поведения состоит в том, что запись на жесткий диск — очень медленная операция.

Считывание из файла, запись в файл осуществляются с помощью функций `fscanf, fprintf`. Отличие от `scanf, printf` состоит в том, что первый аргумент является файлом, из которого мы читаем или в который пишем.

```
#include <stdio.h>

#define TASK_1


#ifdef TASK_1
int main(int argc, char **argv)
{
  #define SIZE 100
  if (argc < 3)
  {
    fprintf(stderr, "Wrong format\n");
    return -3;
  }
```

```
    FILE* fr = fopen(argv[1], "r");
    if (!fr)
    {
        fprintf(stderr, "Can't open file for reading\n");
        return -1;
    }
    FILE* fw = fopen(argv[2], "w");
    if(!fw)
    {
        fprintf(stderr, "Can't open file for writing\n");
        fclose(fr);
        return -2;
    }
    while (!feof(fr))
    {
        char str[SIZE] = { 0 };
        fgets(str, SIZE, fr);
        printf("%s", str);
        fputs(str, fw);
    }
    fclose(fr);
    fclose(fw);
    return 0;
}
#endif

#ifdef TASK_2

int shifr(char* input, char* output)
{
    FILE* fr = fopen(input, "rb");
    if (!fr)
    {
        fprintf(stderr, "Can't open file for reading\n");
        return -1;
```

```
  }
  FILE* fw = fopen(output, "wb");
  if (!fw)
  {
    fprintf(stderr, "Can't open file for writing\n");
    fclose(fr);
    return -2;
  }
  fseek(fr, 0L, SEEK_END);
  long length = ftell(fr);
  fseek(fr, 0L, SEEK_SET);
  unsigned char key = 1;
  for (int i = 0; i < length; ++i)
  {
    unsigned char c = fgetc(fr);
    unsigned char res = c ^ key++;
    fputc(res, fw);
  }
  fclose(fr);
  fclose(fw);
  return 0;
}

int main(int argc, char ** argv)
{
  if (argc < 3)
  {
    fprintf(stderr, "Wrong format\n");
    return -3;
  }
  shifr(argv[1], "shifr.txt");
  shifr("shifr.txt", argv[2]);
}
#endif
```

14 Ссылки и константы

15 Классы

16 Конструкторы и деструкторы

17 Перегрузка

18 Наследование одиночное

19 Наследование множественное

20 Полиморфизм

21 Исключения

22 Шаблоны, их перегрузка и специализация

23 Инстанциирование шаблонов. Вычисления на этапе компиляции

24 Шаблоны, их перегрузка и специализация

25 Итераторы

26 Велосипедируем вектор

27 Велосипедируем мапу

28 Аллокаторы

29 Move-семантика

30 Умные указатели

31 Лямбды