

# **Лекции по C++**

Хлебников Андрей Александрович

16 сентября 2014 г.



The path of the righteous man is beset on all sides by the iniquities of the selfish and the tyranny of evil men.

Blessed is he who, in the name of charity and good will, shepherds the weak through the valley of darkness, for he is truly his brother's keeper and the finder of lost children.

And I will strike down upon thee with great vengeance and furious anger those who attempt to poison and destroy my brothers. And you will know my name is The Lord when I lay my vengeance upon you.

Jules (PULP FICTION, 1994)

# Оглавление

<b>Введение</b>	<b>1</b>
Архитектура ЭВМ	4
Организация памяти	4
Организация исполняемого модуля	4
Типы объектов в C++. POD, non-POD	4
<b>1 Алфавит и основные понятия языка C++. Типы данных.</b>	<b>5</b>
1.1 Алфавит	6
1.2 Литералы	6
1.3 Комментарии	6
1.4 Типы данных языка C++	7
1.5 Операции языка C++	10
<b>2 Операторы языка C++. Структура программы.</b>	<b>15</b>
2.1 Препроцессор	16
2.1.1 Включение файла	16
2.1.2 Макроподстановка	16
2.1.3 Условная компиляция	18
2.1.4 Директива #pragma	19
2.1.5 Директива #pragma pack	20
2.1.6 Директива #pragma section	21
2.1.7 Директива #pragma comment	22
<b>3 Составные типы данных. Массивы. Указатели. Ссылки. Работа с памятью. Функции.</b>	<b>25</b>
3.1 Функции	26
3.2 Типы вызовов функции	29
3.3 Массивы	31
3.4 Составные типы данных	35
3.5 Указатели. Ссылки.	37
3.6 Работа с памятью	48
3.7 Практическая работа	51
<b>4 Файловые потоки. Стандартная библиотека C</b>	<b>55</b>
4.1 Файловые потоки	56
4.2 Стандартная библиотека C	57
4.2.1 Удаление файла	57
4.2.2 Переименование файла	58
4.2.3 Заккрытие файла	58
4.2.4 Сброс буфера	59

4.2.5	Открытие файла . . . . .	59
4.2.6	Связывание потока . . . . .	60
4.2.7	Позиция в файле . . . . .	61
4.2.8	Чтение/Запись . . . . .	65
4.2.9	Индикаторы . . . . .	67
4.2.10	Форматированный ввод/вывод . . . . .	69
4.3	Практическая работа . . . . .	74
<b>5</b>	<b>Потоки ввода/вывода C++ . . . . .</b>	<b>75</b>
	<b>Лабораторные работы . . . . .</b>	<b>77</b>
	Лабораторная работа №1 . . . . .	78
	Лабораторная работа №2 . . . . .	79
	Лабораторная работа №3 . . . . .	80
	Лабораторная работа №4 . . . . .	81
	<b>Курсовая работа . . . . .</b>	<b>83</b>
	Задание . . . . .	85
	Дополнение к реализации . . . . .	86
	<b>Приложение . . . . .</b>	<b>121</b>
	Вопросы к зачету . . . . .	121
	Правила оформления практических заданий . . . . .	124
	Пример оформленного задания . . . . .	124
	Правила оформления лабораторной работы . . . . .	124
	Правила оформления курсовой работы . . . . .	124
	Пример оформленной курсовой работы . . . . .	124
	Правила оформления кода . . . . .	124
	Графики . . . . .	130
	Таблица ASCII . . . . .	132

# Введение

Среди современных языков программирования язык С является одним из наиболее распространенных. Язык С универсален, однако наиболее эффективно его применение в задачах системного программирования — разработке трансляторов, операционных систем, инструментальных средств. Язык С хорошо зарекомендовал себя эффективностью, лаконичностью записи алгоритмов, логической стройностью программ. Во многих случаях программы, написанные на языке С, сравнимы по скорости с программами, написанными в Ассемблере, при этом они более наглядны и просты в сопровождении.

Язык С имеет ряд существенных особенностей, которые выделяют его среди других языков программирования. В значительной степени на формирование идеологии языка повлияла цель, которую ставили перед собой его создатели — обеспечение системного программиста удобным инструментальным языком, который мог бы заменить Ассемблер. В результате появился язык программирования высокого уровня, обеспечивающий необычайно легкий доступ к аппаратным средствам компьютера. С одной стороны, как и другие современные языки высокого уровня, язык С поддерживает полный набор конструкций структурного программирования, модульность, блочную структуру программы. С другой стороны, язык С имеет ряд низкоуровневых черт.

Перечислим некоторые особенности языка С.

В языке С реализован ряд операций низкого уровня. Некоторые из таких операций напрямую соответствуют машинным командам, например, поразрядные операции или операции  $++$  и  $--$ . Базовые типы данных языка С отражают те же объекты, с которыми приходится иметь дело в программе на Ассемблере — байты, машинные слова и т.д. Несмотря на наличие в языке С развитых средств построения составных объектов (массивов и структур), в нем практически отсутствуют средства для работы с ними как с единым целым. Язык С поддерживает механизм указателей на переменные и функции. Указатель — это переменная, предназначенная для хранения машинного адреса некоторой переменной или функции. Поддерживается арифметика указателей, что позволяет осуществлять непосредственный доступ и работу с адресами памяти практически так же легко, как на Ассемблере. Использование указателей позволяет создавать высокоэффективные программы, однако требует от программиста особой осторожности. Как никакой другой язык программирования высокого уровня, язык С «доверяет» программисту. Даже в таком существенном вопросе, как преобразование типов данных, налагаются лишь незначительные ограничения. Однако это также требует от программиста осторожности и самоконтроля. Несмотря на эффективность и мощность конструкция языка С, он относительно мал по объему. В нем отсутствуют встроенные операторы ввода/вывода, динамического распределения памяти, управления процессами и т.п., однако в системное окружение языка С входит библиотека стандартных функций, в которой реализованы подобные действия. Язык С++ — это язык программирования общего назначения, цель которого — сделать работу серьезных программистов более приятным занятием. За исключением несущественных деталей, язык С++ является надмножеством языка С. Помимо возможностей, предоставляемых языком С, язык С++ обеспечивает гибкие и эффективные средства определения новых типов.

Язык программирования служит двум взаимосвязанным целям: он предоставляет программисту инструмент для описания подлежащих выполнению действий и набор концепций, которыми оперирует программист, обдумывая, что можно сделать. Первая цель в идеале требует языка, близкого к компьютеру, чтобы все важные элементы компьютера управлялись просто и эффективно способом, достаточно очевидным для программиста. Язык С создавался на основе именно от этой идеи. Вторая цель в идеале требует языка, близкого к решаемой задаче, чтобы концепции решения могли быть выражены понятно и непосредственно. Эта идея привела к пополнению

языка С свойствами, превратившими его в язык С++.

Ключевое понятие в языке С++ — класс. Классы обеспечивают сокрытие информации, гарантированную инициализацию данных, неявное преобразование определяемых пользователем типов, динамическое определение типа, контроль пользователя над управлением памятью и механизм перегрузки операторов. Язык С++ предоставляет гораздо лучшие, чем язык С, средства для проверки типов и поддержки модульного программирования. Кроме того, язык содержит усовершенствования, непосредственно не связанные с классами, такие как: символические константы, встраивание функций в место вызова, параметры функций по умолчанию, перегруженные имена функций, операторы управления свободной памятью и ссылки. Язык С++ сохраняет способность языка С эффективно работать с аппаратной частью на уровне битов, байтов, слов, адресов и т.д. Это позволяет реализовывать пользовательские типы с достаточной степенью эффективности.

**Архитектура ЭВМ**

**Организация памяти**

**Организация исполняемого модуля**

**Типы объектов в C++. POD, non-POD**



## **Глава 1**

# **Алфавит и основные понятия языка C++. Типы данных.**

## 1.1 Алфавит

Множество символов языка C включает:

- прописные буквы латинского алфавита;
- строчные буквы латинского алфавита;
- арабские цифры;
- разделители: , . ; : ? !

Остальные символы могут быть использованы только в символьных строках, символьных константах и комментариях. Язык C++ различает большие и маленькие буквы, таким образом, `name` и `Name` — разные идентификаторы.

## 1.2 Литералы

Литералы в языке C++ могут быть целые, вещественные, символьные и строковые.

- Целые:
  - десятичные: 10, 132, -32179;
  - восьмеричные (предваряются символом 0): 010, 0204, -076663;
  - шестнадцатеричные (предваряются символами 0x): 0xA, 0x84, 0x7db3.
- Вещественные: 15.75, 1.575e1, .75, -.125
- Символьные: 'a', 'e', '.', '?', '2'.
- Строковые: "строка".

## 1.3 Комментарии

Комментарий — это последовательность символов, которая игнорируется компилятором языка C++. Комментарий имеет следующий вид: `/*<символы>*/`. Комментарии могут занимать несколько строк, но не могут быть вложенными. Кроме того, часть строки, следующая за символами `//`, также рассматривается как комментарий.

## 1.4 Типы данных языка C++

Имя	Размер	Представляемые значения	Диапазон
bool	1 байт	логические	false, true
(signed) char	1 байт	символы и целые числа	от -128 до 127
wchar_t	2 байта	символы Unicode	от 0 до 65535
(signed) short int	2 байта	целые числа	от -32768 до 32767
(signed) int	4 байта	целые числа	от -2147483648 до 2147483647
(signed) long long int	8 байт	целые числа	от -9,223,372,036,854,775,808 до 9,223,372,036,854,775,807
(signed) __int64 (MS)	8 байт		
unsigned char	1 байт	символы и целые числа	от 0 до 255
unsigned short int	2 байта	целые числа	от 0 до 65535
unsigned int	4 байта	целые числа	от 0 до 4294967295
(unsigned) long long int	8 байт	целые числа	от 0 до 18,446,744,073,709,551,615
(unsigned) __int64 (MS)	8 байт		
float	4 байта	вещественные числа	от 1.175494351e-38 до 3.402823466e+38
double	8 байт	вещественные числа	от 2.2250738585072014e-308 до 1.7976931348623158e+308

В языке C++ также существуют перечислимый тип — `enum`, который является подмножеством целого типа, и пустой тип — `void`, который имеет специальное назначение. Он используется для объявления функций, которые не возвращают никакого значения, а также для объявления указателей на значение типа `void`. Такие указатели могут быть преобразованы к указателям на любой другой тип.

В языке C++ можно объявлять структуры и так называемые объединения.

В языке C++ нет специальных типов для массивов и строк, которые представляются массивом символов.

В языке C не существовало логического типа. Логические значения представлялись данными целого типа, при этом значение 0 соответствовало логическому значению ложь, а все остальные целые значения соответствовали логическому значению истина. В языке C++ сохранена данная логика. По определению, `true` имеет значение 1 при преобразовании к целому типу, а `false` — значение 0. И наоборот, целые можно неявно преобразовать в логические значения: при этом ненулевые целые преобразуются в `true`, а ноль — в `false`. В любом месте, где требуется логическое значение, может стоять целочисленное выражение. В арифметических и логических выражениях логические значения преобразуются в целые, операции выполняются над преобразованными величинами.

Указатель можно неявно преобразовать в логическое значение, при этом ненулевой указатель принимает значение `true`, нулевой — `false`.

Такой подход позволяет вместо логической и целочисленной переменных объявлять только целочисленную, при этом значение переменной, равное 0, говорит об отсутствии некоторого признака у объекта, а остальные значения говорят о его наличии, и при этом несут какую-либо дополнительную информацию.

При выполнении бинарных операций производятся преобразования по умолчанию для приведения операндов к одному и тому же типу, который потом используется как тип результата:

- если один из операндов имеет тип `long double`, другой тоже преобразуется в `long double`;
  - иначе, если один операнд имеет тип `double`, то второй операнд преобразуется к типу `double`;
  - иначе, если один операнд имеет тип `float`, то второй операнд преобразуется к типу `float`;
  - иначе над обоими операндами производится интегральное продвижение, а именно: значения типов `char`, `signed char`, `unsigned char`, `short int` и `unsigned short int` преобразуются в `int`, если `int` может представить все значения исходных типов, в противном случае они преобразуются в `unsigned int`;
  - `bool` преобразуется в `int`.
- затем если один операнд имеет тип `unsigned long`, то второй операнд преобразуется к типу `unsigned long`;
  - иначе, если один из операндов относится к типу `long int`, а другой к типу `unsigned int`, то если `long int` может представить все значений типа `unsigned int`, `unsigned int` преобразуется в `long int`, иначе оба операнда преобразуются в `unsigned long int`;

- иначе, если один операнд имеет тип `long int`, то второй операнд преобразуется к типу `long int`;
- иначе, если один операнд имеет тип `unsigned int`, то второй операнд преобразуется к типу `unsigned int`;
- иначе оба операнда имеют тип `int`.

В языке C++ нет операций преобразования между символом и кодом символа, т.к. в оперативной памяти символ и так хранится в виде его кода. Поэтому можно к переменной, хранящей символ, прибавить 1 и получить следующий символ.

## 1.5 Операции языка C++

Данная таблица описывает операции языка C++. Операции разделены на группы, расположенные в порядке убывания приоритета операций.

Знак операции	Наименование	Ассоциативность
::	Разрешение области видимости	Слева направо
() [] . -> ++ -- static_cast dynamic_cast reinterpret_cast const_cast	Первичные Постфиксный инкремент и декремент Преобразование с проверкой во время компиляции Преобразование с проверкой во время выполнения Преобразование без проверки Константное преобразование	Слева направо
- ~ ! * & ++ -- sizeof (<тип>) <выражение> new delete	Унарные Префиксный инкремент и декремент Вычисление размера Приведение типа Выделение памяти Освобождение памяти	Справа направо
. * -> *	Выбор члена класса	
* / %	Мультипликативные	Слева направо
+ -	Аддитивные	Слева направо
<< >>	Сдвиг	Слева направо
< > <= >=	Отношение	Слева направо
== !=	Отношение	Слева направо
&	Поразрядное И	Слева направо
⊖	Поразрядное исключающее ИЛИ	Слева направо
	Поразрядное ИЛИ	Слева направо
&&	Логическое И	Слева направо
	Логическое ИЛИ	Слева направо

?:	Условная операция	Слева направо
= *= /= %= += -= <<= >>= &= =□	Простое и составное присваивания	Слева направо
throw	Генерация исключения	Слева направо
,	Операция последовательного вычисления	Слева направо

- `::` — операция разрешения области видимости. При повторном объявлении имени во вложенном блоке или классе предыдущие объявления оказываются скрытыми. Однако скрытое имя члена класса можно использовать, квалифицировать его именем класса при помощи операции разрешения области видимости. Скрытое глобальное имя можно использовать, если квалифицировать его унарной операцией разрешения области видимости;
- `()` — выражение в скобках (используется для изменения порядка вычисления) или вызов функции;
- `[]` — индексное выражение, используется для работы с массивами;
- `.` и `->` — выбор элемента, используются при работе с классами, структурами и объединениями;
- Операции `static_cast` (преобразование с проверкой во время компиляции), `dynamic_cast` (преобразование с проверкой во время выполнения), `reinterpret_cast` (преобразование без проверки), `const_cast` (константное преобразование) осуществляют различные преобразования типов. Они имеют следующий синтаксис: операция<новый тип>(выражение). Угловые скобки являются элементом синтаксиса. Операция `static_cast` осуществляет преобразование родственных типов, например, указателя на один тип к указателю на другой тип из той же иерархии классов, целый тип в перечисление или тип с плавающей точкой в интегральный. Операция `reinterpret_cast` управляет преобразованиями между несвязанными типами, например, целых в указатели или указателей в другие (несвязанные) указатели. Такое различие позволяет компилятору осуществлять минимальную проверку типов при использовании `static_cast`, а программисту — легче обнаружить опасные преобразования, представляемые `reinterpret_cast`. Преобразование `dynamic_cast` выполняется и проверяется на этапе выполнения. Преобразование `const_cast` аннулирует действие модификатора `const`.
- `-` — унарный минус;
- `~` — обратный код;
- `!` — логическое отрицание;
- `*` — косвенная адресация;
- `&` — адресация;
- Операции `++` и `--` инкрементируют (увеличивают на 1) и декрементируют (уменьшают на 1) свой операнд. Операнд должен иметь целый, вещественный тип или быть указателем. Операции инкремента и декремента могут записываться как перед своим операндом (префиксная форма записи), так и после него (постфиксная форма записи). При префиксной форме записи операнд сначала инкрементируется или декрементируется, а затем его новое значение участвует в дальнейшем вычислении выражения, содержащего данную операцию. При постфиксной форме записи операнд инкрементируется или декрементируется лишь после того, как его старое значение участвует в вычислении выражения. Таким образом, результатом операций инкремента и декремента является либо новое, либо старое значение операнда. Например, если переменная `i = 0`, то выражение `a[++i] = 1` меняет элемент `a[1]`, а выражение `a[i++] = 1`



меняет элемент `a[0]`. В обоих случаях переменная `i` получает новое значение, равное 1.

- `sizeof` — вычисление размера в байтах переменной или типа;
- Операция приведения типа записывается следующим образом: `(<новый тип>) <выражение>`. Например, `(long int)n` приводит переменную `n` к типу `long int`. При преобразовании типов надо помнить, что при преобразовании между знаковыми/беззнаковыми значениями и при преобразовании от типа с большей размерностью к типу с меньшей размерностью могут возникнуть ошибки. Более безопасным способом преобразования типов является использование операций `static_cast`, `dynamic_cast`, `reinterpret_cast` и `const_cast`.
- `%` — остаток от деления;
- В языке C++ имеется одна тернарная операция — условная операция. Она имеет следующий синтаксис: `<операнд 1> ? <операнд 2> : <операнд 3>`. Если `<операнд 1>` имеет ненулевое значение, то вычисляется `<операнд 2>` и результатом условной операции является его значение. Если же `<операнд 1>` равен нулю, то вычисляется `<операнд 3>` и результатом является его значение. В любом случае вычисляется только один из операндов, `<операнд 2>` или `<операнд 3>`, но не оба;
- Простое присваивание. Операция простого присваивания обозначается знаком `=`. Значение правого операнда присваивается левому операнду. Операция вырабатывает результат, который может быть далее использован в выражении. Результатом операции является присвоенное значение. Например, выражение `a = b = c = 0` присваивает всем переменным значение 0, а в результате вычисления выражения `a = (b = 3) + (c = 5)` переменная `c` будет иметь значение 5, переменная `b` будет иметь значение 3, и переменная `a` будет иметь значение 8;
- Составное присваивание. Операция составного присваивания состоит из простой операции присваивания, скомбинированной с какой-либо другой бинарной операцией. При составном присваивании вначале выполняется действие, специфицированное бинарной операцией, а затем результат присваивается левому операнду. Оператор `n += 5` эквивалентен оператору `n = n + 5`, но при этом первый оператор легче для понимания и выполняется быстрее.
- Операция последовательного вычисления, обычно используется для вычисления нескольких выражений в ситуациях, где по синтаксису допускается только одно выражение. Однако, запятая, разделяющая параметры функции, не является операцией последовательного вычисления.

Порядок вычислений подвыражений внутри выражений не определён. В частности, не стоит предполагать, что выражения вычисляются слева направо.

```
int x = f(2) + g(3);
```

При отсутствии ограничений на порядок вычислений можно сгенерировать более качественный код. Однако отсутствие ограничений на порядок вычислений может привести к неопределённым результатам.

Логические операции И и ИЛИ, условная операция и операция последовательного вычисления гарантируют определенный порядок вычисления своих операндов. Условная операция вычисляет сначала свой первый операнд, а затем, в зависимости от его значения, либо второй, либо третий операнд. Логические операции также обеспечивают вычисление своих операндов слева направо, причём логические операции вычисляют минимальное число операндов, необходимое для определения результата выражения. Таким образом, второй операнд выражения может вообще не вычисляться. Операция последовательного вычисления обеспечивает вычисление своих операндов по очереди, слева направо. Обратите внимание, что запятая в качестве указателя последовательности логически отличается от запятой, используемой в качестве разделителя параметров при вызове функций.

```
f1( v[i], i++ );  
f2( (v[i], i++) );
```

Вызов функции f1 осуществляется с двумя параметрами `v[i]` и `i++`, и порядок вычисления параметров не определён. Расчет на определённый порядок вычисления параметров является исключительно плохим стилем и приводит к непредсказуемому поведению программы. Вызов функции f2 имеет один параметр — последовательность выражений, разделённых запятой. Порядок вычисления гарантирован, и вызов эквивалентен `f2(i++)`.

## **Глава 2**

# **Операторы языка C++. Структура программы.**

## 2.1 Препроцессор

### 2.1.1 Включение файла

Включение файлов<sup>1</sup> (помимо других полезных вещей) позволяет легко управлять наборами директив `#define` и объявлений. Любая строка вида

```
#include "filename"
```

или

```
#include <filename>
```

заменяется содержимым файла с именем `filename`. Если имя файла заключено в двойные кавычки, то, как правило, файл ищется среди исходных файлов программы; если такового не оказалось или имя файла заключено в угловые скобки `<` и `>`, то поиск осуществляется по определяемым реализацией правилам. Включаемый файл сам может содержать в себе строки `#include`. Часто исходные файлы начинаются с нескольких строк `#include`, ссылающихся на файлы, содержащие общие директивы `#define`, объявления `extern` или прототипы нужных библиотечных функций из заголовочных файлов вроде `<stdio.h>`. (Строго говоря, эти включения не обязательно являются файлами; технические детали того, как осуществляется доступ к заголовкам, зависят от конкретной реализации.) Директива `#include` — хороший способ собрать вместе объявления большой программы. Он гарантирует, что все исходные файлы будут пользоваться одними и теми же определениями и объявлениями переменных, благодаря чему предотвращаются особенно неприятные ошибки. Естественно, при внесении изменений во включаемый файл все зависимые от него файлы должны перекомпилироваться.

### 2.1.2 Макроподстановка

Директива макроподстановки имеет вид:

```
#define имязамещающий \текст_
```

Макроподстановка используется для простейшей замены: во всех местах, где встречается имя, вместо него будет помещен замещающий\_текст. Имена в `#define` задаются по тем же правилам, что и имена обычных переменных. Замещающий текст может быть произвольным. Обычно замещающий текст целиком помещается в строке, в которой расположено слово `#define`, однако длинные определения можно разбивать на несколько строк, поставив в конце каждой продолжаемой строки обратную наклонную черту `\`. Область видимости имени, определенного директивой `#define`, простирается от определения до конца файла. В определении макроподстановки могут использоваться предшествующие ему макроопределения. Подстановка осуществляется только для тех имен, которые расположены вне текстов заключенных в кавычки и не являются частью другого слова. Например, если `YES` определено с помощью директивы `#define`, то никакой подстановки в `printf("YES")` или в `YESMAN` выполнено не будет. Любое имя можно определить с произвольным замещающим текстом. Например,

```
#define forever for(;;) /* бесконечныйцикл*/
```

<sup>1</sup>Текст позаимствован с ресурса [http://netlib.narod.ru/library/book0003/ch04\\_11.htm](http://netlib.narod.ru/library/book0003/ch04_11.htm)

определяет новое слово `forever` для бесконечного цикла. Можно определить макрос с аргументами, чтобы замещающий текст варьировался в зависимости от задаваемых параметров. Например, определим `max` следующим образом:

```
#define max(A, B) ((A) > (B) ? (A) : (B))
```

Хотя обращения к `max` выглядят как обычные обращения к функции, они будут вызывать только текстовую замену. Каждый формальный параметр (в данном случае `A` и `B`) будет заменяться соответствующим ему аргументом. Так, строка

```
x = max(p + q, r + s);
```

будет заменена на строку

```
x = ((p + q) > (r + s) ? (p + q) : (r + s));
```

Поскольку аргументы просто подставляются в текст, указанное определение `max` подходит для данных любого типа, так что не нужно писать различные варианты `max` для данных разных типов, как это было бы в случае определения функции. Если вы внимательно проанализируете работу `max`, то обнаружите некоторые подводные камни. Выражения вычисляются дважды, и если они вызывают побочный эффект (из-за инкрементных операций или функций ввода-вывода), это может привести к нежелательным последствиям. Например,

```
max(i++, j++) /* НЕВЕРНО*/
```

вызовет увеличение `i` и `j` дважды. Кроме того, следует позаботиться о скобках, чтобы обеспечить нужный порядок вычислений. Задумайтесь, что случится, если при определении

```
#define square(x) x * x /* НЕВЕРНО*/
```

вызвать `square(z + 1)`. Тем не менее макросы имеют свои достоинства. Один из примеров можно найти в файле `<stdio.h>`, где `getchar` и `putchar` часто реализуют с помощью макросов, чтобы избежать расходов времени на вызов функции для каждого обрабатываемого символа. Функции в `<ctype.h>` обычно также реализуются с помощью макросов. Действие `#define` можно отменить с помощью директивы `#undef`:

```
#undef getchar
```

```
int getchar(void) { ... }
```

Как правило, это делается, чтобы заменить макроопределение настоящей функцией с тем же именем. Имена формальных параметров не заменяются, если встречаются в заключенных в кавычки строках. Однако, если в замещающем тексте перед формальным параметром стоит знак `#`, этот параметр будет заменен на аргумент, заключенный в кавычки. Это можно сочетать с конкатенацией строк, например, чтобы создать макрос отладочного вывода:

```
#define dprint(expr) printf(#expr " = %g\n", expr)
```

Обращение к

```
dprint(x/y);
```

развернется в

```
printf("x/y" " = %g\n", x/y);
```

а в результате конкатенации двух соседних строк, которая будет автоматически выполнена компилятором, получим

```
printf("x/y = %g\n", x/y);
```

Внутри фактического аргумента каждый знак " заменяется на \", а каждая обратная наклонная черта \ на \\. так что в результате подстановки получается правильная символьная константа. Оператор ## позволяет конкатенировать аргументы в макрорасширениях. Если в замещающем тексте параметр соседствует с ##, то он заменяется соответствующим ему аргументом, а оператор ## и окружающие его символы-разделители выбрасываются. Например, в макроопределении paste конкатенируются два аргумента

```
#define paste(front, back) front ## back
```

так что запись paste(name, 1) будет заменена на name1.

### 2.1.3 Условная компиляция

Работой препроцессора можно управлять с помощью условных инструкций. Они представляют собой средство для выборочного включения того или иного текста программы в зависимости от значения условия, вычисляемого во время компиляции. Вычисляется константное целое выражение, заданное в строке #if. Это выражение не должно содержать операторы sizeof, приведения типов и констант из перечислений enum. Если оно имеет ненулевое значение, то будут включены все последующие строки вплоть до ближайшей директивы #endif, #elif, или #else. (Директива препроцессора #elif действует как else if.) Выражение defined(имя) в #if равно 1, если имя было определено, и 0 в противном случае. Например, чтобы застраховаться от повторного включения заголовочного файла hdr.h, его можно оформить следующим образом:

```
#if !defined(HDR)
#define HDR

/* здесь содержимое hdr.h */

#endif
```

При первом включении файла hdr.h будет определено имя HDR, а при последующих включениях препроцессор обнаружит, что имя HDR уже определено, и перескочит сразу на #endif. Этот прием может оказаться полезным, когда нужно избежать многократного включения одного и того же файла. Если им пользоваться систематически, то в результате каждый заголовочный файл будет сам включать заголовочные файлы, от которых он зависит, освободив от этого занятия пользователя. Вот пример цепочки проверок имени SYSTEM, позволяющей выбрать нужный файл для включения:

```
#if SYSTEM == SYSV
#define HDR "sysv.h"
#elif SYSTEM == BSD
#define HDR "bsd.h"
#elif SYSTEM == MSDOS
#define HDR "msdos.h"
#else
```

```
#define HDR "default.h"
#endif
#include HDR
```

Инструкции `#ifdef` и `#ifndef` специально предназначены для проверки того, определено или нет заданное в них имя. И следовательно, первый пример, приведенный выше для иллюстрации `#if`, можно записать и в таком виде:

```
#ifndef HDR
#define HDR

/* здесь содержимое hdr.h */

#endif
```

### 2.1.4 Директива `#pragma`

Директивы `pragma`<sup>2</sup> определяют функции компилятора для конкретного компьютера или операционной системы.

```
#pragma string
```

Каждая реализация C и C++ поддерживает некоторые функции, уникальные для хост-компьютера или операционной системы. Например, некоторые программы должны осуществлять точный контроль над областями памяти, в которых размещаются данные, или управлять способом получения параметров некоторыми функциями. Директивы `#pragma` позволяют компилятору использовать особенности конкретного компьютера или операционной системы, сохраняя при этом совместимость языков C и C++. Директивы `pragma` характерны для конкретного компьютера или операционной системы по определению и обычно отличаются для каждого компилятора. Директивы `pragma` можно использовать в условных операторах для обеспечения новой функциональности препроцессора или для предоставления компилятору сведений, определенных реализацией.

`string` — это последовательность символов, которые предоставляют определенную инструкцию компилятора и аргументы, если таковые имеются. Символ решетки (`#`) должен быть первым отличным от пробела символом в строке, которая содержит директиву `pragma`; символы пробела могут разделять знак числа и слово `pragma`. После `#pragma` введите любой текст, который преобразователь может проанализировать как токены предварительной обработки. Аргумент `#pragma` подлежит расширению макроса. Если компилятор обнаруживает нераспознаваемую директиву `pragma`, он выдает предупреждение и продолжает компиляцию.

Компиляторы Microsoft C и C++ распознают следующие директивы `pragma`.

<code>alloc_text</code>	<code>auto_inline</code>	<code>bss_seg</code>
<code>check_stack</code>	<code>code_seg</code>	<code>comment</code>
<code>component</code>	<code>conform</code>	<code>const_seg</code>
<code>data_seg</code>	<code>deprecated</code>	<code>detect_mismatch</code>
<code>fenv_access</code>	<code>float_control</code>	<code>fp_contract</code>
<code>function</code>	<code>hdrstop</code>	<code>include_alias</code>
<code>init_seg</code>	<code>inline_depth</code>	<code>inline_recursion</code>

---

<sup>2</sup><http://msdn.microsoft.com/ru-ru/library/d9x1s805.aspx>

<code>intrinsic</code>	<code>loop</code>	<code>make_public</code>
<code>managed</code>	сообщение	<code>omp</code>
<code>once</code>	<code>optimize</code>	<code>pack</code>
<code>pointers_to_members</code>	<code>pop_macro</code>	<code>push_macro</code>
<code>region, endregion</code>	<code>runtime_checks</code>	<code>section</code>
<code>setlocale</code>	<code>strict_gs_check</code>	<code>unmanaged</code>
<code>vtordisp</code>	<code>warning</code>	

### 2.1.5 Директива `#pragma pack`

Задаёт выравнивание упаковки для членов структуры, объединения и класса.

```
#pragma pack( [ show ] | [ push | pop ] [, identifier ] , n )
```

Директива `pack` обеспечивает контроль на уровне объявления данных. В этом состоит отличие от параметра компилятора `/Zp`, который предоставляет контроль только на уровне модуля. Директива `pack` действует на первое объявление `struct`, `union` или `class` после этой директивы `#pragma`. Директива `pack` не действует на определения. При вызове директивы `pack` без аргументов для параметра `n` задается значение, указанное в параметре компилятора `/Zp`. Если этот параметр компилятора не указан, по умолчанию используется значение 8. При изменении выравнивания структуры она может занимать меньше места в памяти, но возможно снижение производительности или даже возникновение аппаратного исключения для невыровненного доступа. Поведение этого исключения можно изменить с помощью директивы `SetErrorMode`.

- `show`(необязательно) — Отображает текущее байтовое значение выравнивания упаковки. Значение отображается в предупреждении.
- `push`(необязательно) — Помещает текущее значение выравнивания упаковки во внутренний стек компилятора и задает для текущего выравнивания упаковки значение `n`. Если значение `n` не указано, текущее значение выравнивания упаковки не помещается в стек.
- `pop`(необязательно) — Удаляет запись из вершины внутреннего стека компилятора. Если значение `n` не указано вместе с `pop`, то новым значением выравнивания упаковки становится значение упаковки, связанное с результирующей записью в вершине стека. Если значение `n` указано (например, `#pragma pack(pop, 16)`), `n` становится новым значением выравнивания упаковки. Если извлечение из стека производится вместе с идентификатором `identifier` (например, `#pragma pack(pop, r1)`), из стека извлекаются все записи, пока не будет найдена запись `identifier`. Эта запись извлекается из стека, и новым значением выравнивания упаковки становится значение упаковки, связанное с результирующей записью в вершине стека. Если при извлечении из стека с идентификатором `identifier` этот идентификатор не найден ни в одной из записей стека, директива `pop` игнорируется.
- `identifier`(необязательно) — При использовании с директивой `push` присваивает имя записи во внутреннем стеке компилятора. При использовании с директивой



pop записи из внутреннего стека извлекаются до тех пор, пока не будет удален идентификатор `identifier`; если идентификатор `identifier` во внутреннем стеке не найден, ничего не извлекается.

- `n` (необязательно) — Указывает значение (в байтах), используемое для упаковки. Если для модуля не задан параметр компилятора `/Zp`, по умолчанию для `n` используется значение 8. Допустимые значения: 1, 2, 4, 8 и 16. Выравнивание члена будет производиться по границе, кратной значению `n` или кратной размеру члена, в зависимости от того, какое из значений меньше.

Значение `#pragma pack(pop, identifier, n)` не определено.

```
/* pragma_directives_pack.c */
#include <stddef.h> /** offsetof */
#include <stdio.h>

struct S {
    int i; /* size 4*/
    short j; /* size 2*/
    double k; /* size 8*/
};

#pragma pack(2)
struct T {
    int i;
    short j;
    double k;
};

int main() {
    printf("%d ", offsetof(struct S, i));
    printf("%d ", offsetof(struct S, j));
    printf("%d - ", offsetof(struct S, k));

    printf("%d ", offsetof(struct T, i));
    printf("%d ", offsetof(struct T, j));
    printf("%d\n", offsetof(struct T, k));
}
```

Результат

```
0 4 8 - 0 4 6
```

## 2.1.6 Директива `#pragma section`

Создает раздел в OBJ-файле.

```
#pragma section( "section-name" [, attributes] )
```

Термины сегмент и раздел в этом разделе взаимозаменяемы. После определения раздел остается допустимым для остальной части компиляции. Однако следует использовать `__declspec(allocate)`, так как иначе никакие данные не будут помещены в раздел.

- **section-name** — обязательный параметр, который будет именем раздела. Имя не должно конфликтовать со стандартными именами раздела. Список имен, которые не следует использовать при создании раздела, см. в разделе /SECTION.
- **attributes** — необязательный параметр, состоящий из одного или нескольких разделенных запятыми атрибутов, которые требуется присвоить разделу. Ниже перечислены возможные **attributes**.

**read** (чтение) — Позволяет выполнять операции чтения данных.

**write** (запись) — Позволяет выполнять операции записи данных.

**execute** — Позволяет выполнять код.

**shared** — Предоставляет совместный доступ к разделу всем процессам, загружающим образ.

**norpage** — Отмечает раздел как невыгружаемый; используются для драйверов устройств win32.

**nocache** — Отмечает раздел как некешируемый; используются для драйверов устройств win32.

**discard** — Отмечает раздел как удаляемый; используются для драйверов устройств win32.

**remove** — Отмечает раздел как нерезидентный; только драйверы виртуальных устройств (VxD).

Если не задать атрибуты, раздел будет иметь атрибуты чтения и записи.

В следующем примере первая инструкция определяет раздел и его атрибуты.

Целое число *j* не помещается в *mysec*, поскольку оно не было объявлено с `__declspec(allocate(j))` переходит в раздел данных. Целое число *i* переходит в *mysec* как результат атрибута класса хранения `__declspec(allocate)`.

```
/* pragma_section.cpp */
#pragma section("mysec", read, write)
int j = 0;

__declspec(allocate("mysec"))
int i = 0;

int
main()
{ }
```

### 2.1.7 Директива #pragma comment

Вставляет запись комментария в объектный или исполняемый файл.

```
#pragma comment( comment-type [, "commentstring"] )
```

**comment-type** обозначает один из предопределенных идентификаторов (см. ниже), которые задают тип записи комментария. Необязательный параметр **commentstring** обозначает строковый литерал, который содержит дополнительную информацию (для некоторых типов комментариев). Поскольку параметр **commentstring** обозначает

строковый литерал, он соблюдает все правила, действующие для строковых литералов в отношении `escape`-символов, внедренных кавычек (") и конкатенации.

- `compiler` — Задает в объектном файле имя и номер версии компилятора. Эта запись комментария игнорируется компоновщиком. Если для этого типа записи будет указан параметр `commentstring`, компилятор выведет предупреждение.
- `exestr` — Задает в объектном файле `commentstring`, которая во время компоновки помещается в исполняемый файл. Эта строка не загружается в память вместе с исполняемым файлом, однако ее можно обнаружить при помощи программы, которая находит в файлах печатаемые строки. Записи комментариев этого типа позволяют, в частности, вставлять в исполняемый файл информацию о номере версии и т. д. Использовать параметр `exestr` не рекомендуется; в следующих выпусках он будет удален. Компоновщик не обрабатывает эту запись комментария.
- `lib` — Задает в объектном файле запись поиска библиотеки. Этот тип комментария должен сопровождаться `commentstring` с именем библиотеки, которую должен найти компоновщик. В нем также можно указать путь к ней. Имя библиотеки указывается в объектном файле после записей поиска по библиотекам по умолчанию. Компоновщик ищет эту библиотеку точно так же, как если бы она была указана в командной строке (если библиотека не была задана при помощи параметра `/NODEFAULTLIB`). В один и тот же исходный файл можно вставить несколько записей поиска библиотеки. В объектном файле они будут располагаться в том же порядке, что и в исходном.  
  
Если для вас важно, в каком порядке расположены и добавленные библиотеки и библиотека по умолчанию, задайте при компиляции ключ `/Z1`. В этом случае в объектный модуль не будет вставлено имя библиотеки по умолчанию. Далее можно вставить еще одну директиву `#pragma comment` и с ее помощью добавить имя библиотеки по умолчанию уже после добавленной библиотеки. Библиотеки, добавленные при помощи этих директив, будут находиться в объектном модуле в том же порядке, в каком они указаны в исходном коде.
- `linker` — Задает параметр компоновщика в объектном файле. Благодаря этому параметр компоновщика можно не передавать из командной строки и не указывать в среде разработки, а задать непосредственно в комментарии. Например, в нем можно задать параметр `/INCLUDE`, чтобы принудительно включить символ:

```
#pragma comment(linker, "/INCLUDE:__mySymbol")
```

Идентификатору компоновщика могут передаваться только следующие параметры (`comment-type`)

- \* `/DEFAULTLIB`
- \* `/EXPORT`
- \* `/INCLUDE`
- \* `/MANIFESTDEPENDENCY`

- \* /MERGE
- \* /SECTION

- **user** — Вставляет в объектный файл комментарий общего рода. `commentstring` содержит текст комментария. Эта запись комментария игнорируется компоновщиком.

Следующая директива `pragma` указывает компоновщику найти библиотеку `EMAPI.LIB` во время компоновки. Сначала компоновщик ищет ее в текущем рабочем каталоге, а затем по пути, заданном в переменной среды `LIB`.

```
#pragma comment( lib, "emap1" )
```

Следующая директива `pragma` указывает компилятору вставить в объектный файл имя и номер версии компилятора:

```
#pragma comment( compiler )
```

```
#pragma comment( user, "Compiled on " __DATE__ " at " __TIME__ )
```

## **Глава 3**

**Составные типы данных.  
Массивы. Указатели. Ссылки.  
Работа с памятью. Функции.**

### 3.1 Функции

Функция – это группа операторов у которой есть имя. Функции позволяют разбить программу на небольшие части, каждая из которых выполняет какую-то небольшую задачу. Без функций довольно проблематично написать эту программу, в которой более 1000 строк кода.

Обязательными для функции являются два компонента: определение и вызовы.

Определение функции должно располагаться в глобальной области видимости, до начала функции `main`. Рассмотрим пример, простого определения:

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

Определение функции состоит из заголовка и тела. Заголовок функции включает в себя:

#### Тип возвращаемого значения

Почти все функции должны возвращать значения. Тип этого значения указывается в заголовке перед именем функции. Вот несколько примеров заголовков функций:

```
int simple_function_int()
float simple_function_float()
char simple_function_char()
```

В первом случае функция должна вернуть целое число (`int`), во втором – вещественное число (`float`), а в третьем случае – символ (`char`). Возвращаемые значения используются для передачи данных из функции в вызывающее окружение. Вызывающее окружение – это то место, откуда вызывается данная функция, подробнее ниже.

#### Идентификатор или имя функции

Идентификатор (имя) функции задаётся точно так же, как и любой другой идентификатор. В данном примере мы создали функцию с идентификатором `simple_function`.

#### Список аргументов или параметров

Список аргументов функции записывается в круглых скобках после имени функции. В данном примере список аргументов пуст. Список аргументов записывается через запятую. Каждый элемент списка состоит из типа и идентификатора. Рассмотрим пример заголовка функции со списком из двух аргументов:

```
int simple (int a, float b)
```

В скобках мы записали два аргумента: `a` и `b`. У аргумента `a` тип `int`, а у аргумента `b` тип `float`. Аргументы используются, когда в функцию нужно передать какие-либо данные из вызывающего окружения.

#### Тело функции

Тело функции располагается сразу под заголовком и заключено в фигурные скобки. В теле функции может содержаться сколько угодно операторов. Но обязательно должен присутствовать оператор `return`. Оператор `return` возвращает значение:

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

Здесь, `simple_function` всегда будет возвращать 0. Надо признать, что данная функция бесполезна. Напишем функцию, которая принимает из вызывающего окружения два значения, складывает их и возвращает результат в вызывающее окружение. Назовём эту функцию `sum`<sup>1</sup>(сумма):

Листинг 3.1: Не упрощенный вариант

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

Листинг 3.2: Упрощенный вариант

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

В функцию передаётся два аргумента: `a` и `b` типа `int`. В теле функции они используются как обычные переменные (они и являются обычными переменными). Давайте договоримся: снаружи функции, выражения, которые передаются в неё, мы будем называть аргументами, а эти же переменные в теле функции – параметрами. В теле функции определяется переменная `c`. А затем, в эту переменную мы помещаем значение суммы двух параметров. Последняя строчка возвращает значение переменной `c` во внешнее окружение. После ключевого слова `return` нужно указать значение которое будет возвращено. Можно возвращать как простые значения, так и переменные и даже выражения. Например:

```
return 32;
return a;
return b;
return a + b;
```

В последнем случае в вызывающее окружение будет возвращён результат суммы переменных `a` и `b`. Обратите внимание, что оператор `return` не только возвращает значение, но и служит как бы выходом из функции, после него не будет выполнен ни один оператор:

```
return a;
c = a + b; // Isn't executed
```

Благодаря этому, с помощью `return` удобно создавать условия выхода из функций:

```
if (a > 0) {
    return 0;
} else if (a < 0) {
    return 1;
}
```

---

<sup>1</sup> Приведенный код записан не совсем оптимально, выделяется дополнительная переменная для хранения результата. Проще было бы написать 3.2. Но в данном конкретном случае, современный компилятор самостоятельно избавится от временной переменной и выражение 3.1 приведет к 3.2

Здесь, из функции будет возвращено число в зависимости от значения переменной `a`: если `a` больше нуля, то будет возвращён 0, в противном случае – 1.

### Вызов функции

После того как создано определение функции, её можно вызвать.

```
#include <iostream>

int sum (int a, int b)
{
    int c;
    c = a + b;
    return c;
}

int main()
{
    int s = sum(2, 2); // Function call
    std::cout << s;
    return 0;
}
```

В результате выполнения программы, на экран будет выведено: 4.

Вызов функции состоит из идентификатора функции и списка аргументов в круглых скобках. Вот несколько вызовов функции `sum`:

```
int x = 5;
int y = 4;
int z;

sum(0, 1); // 1
sum(x, 2); // 7
sum(x, y); // 9
z = sum(x, y); // z = 9
```

### Вызывающее окружение

То место, откуда вызывается функция, называется вызывающим окружением. Вызывающим окружением функции `sum` является функция `main`, а вызывающим окружением функции `main` является отладчик или операционная система. Функция может обмениваться данными с вызывающим окружением благодаря списку аргументов и возвращаемому значению: вызывающее окружение передаёт данные в функцию с помощью аргументов, а функция передаёт данные в вызывающее окружение с помощью возвращаемого значения. Тип передаваемого в функцию значения должен совпадать с типом указанным в списке аргументов. Нельзя, например, написать вот так<sup>2</sup>:

```
int simple (int a)
{
    return 1;
}

int main ()
{
```

---

<sup>2</sup> Скорее всего будет просто предупреждение компилятора `warning`



```
int b;  
b = simple(0.5);  
return 0;  
}
```

В списке аргументов мы указали тип `int`, а в функцию передаётся вещественное значение `0.5`. Так делать нельзя.

## 3.2 Типы вызовов функции

Соглашение вызова<sup>3</sup> определяет следующие особенности процесса использования подпрограмм:

- Расположение входных параметров подпрограммы и возвращаемых ею значений. Наиболее распространённые варианты:
  - в регистрах;
  - в стеке;
  - в регистрах и стеке.
- Порядок передачи параметров. При использовании для параметров стека определяет, в каком порядке параметры должны быть помещены в стек, при использовании регистров — порядок сопоставления параметров и регистров. Варианты:
  - прямой порядок — параметры размещаются в том же порядке, в котором они перечислены в описании подпрограммы. Преимущество — единообразие кода и записи на языке высокого уровня;
  - обратный порядок — параметры передаются в порядке от конца к началу. Преимущество — при любом количестве параметров на вершине стека после адреса возврата оказывается сначала первый параметр, за ним второй и так далее. Это упрощает реализацию подпрограмм с неопределённым числом параметров произвольных типов.
- Кто возвращает указатель стека на исходную позицию:
  - вызываемая подпрограмма — это сокращает объём команд, необходимых для вызова подпрограммы, поскольку команды восстановления указателя стека записываются только один раз, в конце подпрограммы;
  - вызывающая программа — в этом случае вызов становится сложнее, но облегчается использование подпрограмм с переменным количеством и типом параметров.
- Какой командой вызывать подпрограмму и какой — возвращаться в основную программу. Например, в стандартном режиме x86 подпрограмму можно вызвать через `call near`, `call far` и `pushf/call far` (для возврата применяются соответственно `ret`, `retf`, `iret`).
- Содержимое каких регистров процессора подпрограмма обязана восстановить перед возвратом.

---

<sup>3</sup>Взято из Wikipedia

Соглашения вызова зависят от архитектуры целевой машины и компилятора.

Распространённые соглашения вызова на x86 архитектуре. Во всех нижеперечисленных соглашениях (кроме `cdecl`) подпрограмма обязана обеспечить восстановление перед возвратом значений сегментных регистров процессора, а также регистров ESP и EBP. Значения остальных могут не восстанавливаться. Возвращаемое значение функции хранится в регистре `eax`. Если его размер слишком велик для размещения в регистре, то оно размещается на вершине стека, а значение в регистре `eax` будет указывать на него.

- **`cdecl`** Основной способ вызова для C (отсюда название, сокращение от "c-declaration"). Аргументы передаются через стек, справа налево. Очистку стека производит вызывающая программа. Это основной способ вызова функций с переменным числом аргументов (например, `printf(...)`). Результат функции возвращается через регистр `EAX`, кроме чисел с плавающей точкой — они будут в псевдостеке x87 (в регистре `ST0`).
- **`pascal`** Основной способ вызова для Паскаля, также применялся в Windows 3.x. Аргументы передаются через стек, слева направо. Указатель стека на исходную позицию возвращает подпрограмма. Причём, изменяемые параметры передаются только по ссылке, а у функций неявно создаётся дополнительный первый изменяемый параметр `Result`, через который и возвращается значение.
- **`stdcall`/winapi** Применяется при вызове функций WinAPI. Аргументы передаются через стек, справа налево. Очистку стека производит вызываемая подпрограмма.
- **`fastcall`** Передача параметров через регистры, обычно самая быстрая; если все параметры и промежуточные результаты умецаются в регистрах, манипуляции со стеком вообще не нужны.

`Fastcall` не стандартизирован, поэтому используется только в функциях, которые программа не экспортирует наружу и не импортирует извне.

В компиляторе Borland, для соглашения `__fastcall`, называемого также `register`, параметры передаются слева направо в `eax`, `edx`, `ecx` и, если параметров больше трёх, в стеке, также слева направо. Указатель стека на исходное значение возвращает вызываемая подпрограмма. `Fastcall` Borland применяется по умолчанию в Delphi.

Соглашение `__fastcall` Microsoft, также называемое `__msfastcall`, в 32-разрядной версии компилятора Microsoft, а также компилятора GCC<sup>4</sup>, определяет передачу первых двух параметров слева направо в `ECX` и `EDX`, а остальные параметры передаются справа налево в стеке. Очистку стека производит вызываемая подпрограмма.

- **`safecall`** Обеспечивает более удобный для использования в распространённых языках высокого уровня способ вызова методов интерфейсов при использовании модели COM.

Все методы интерфейсов COM представляют собой функции, возвращающие код завершения типа `HRESULT`, который должен анализироваться в месте вызова.

---

<sup>4</sup>Found in versions 3.4. On the Intel 386, the 'fastcall' attribute causes the compiler to pass the first two arguments in the registers ECX and EDX. Subsequent arguments are passed on the stack. The called function will pop the arguments off the stack. If the number of arguments is variable all arguments are pushed on the stack.

Как правило, это не вполне удобно: большинство используемых с этой технологией языков имеют механизмы обработки исключений, и в них удобнее использовать обычные вызовы функций и процедур, возвращающие прикладные значения, а для обработки ошибок применять исключения. Именно такую возможность предоставляет `safeCall`.

- `thisCall` Используется в компиляторах C++. Обеспечивает передачу аргументов при вызовах методов класса в объектно ориентированной среде. Аргументы передаются через стек, справа налево. Очистку стека производит вызываемая функция, то есть тот же самый `stdCall`. Указатель (`this`) на объект, для которого вызывается метод, записывается в регистр ECX.

### 3.3 Массивы

В большинстве случаев программам необходимо хранить множество значений, например 50 тестовых очков, 100 названий книг или 1000 имен файлов. Если вашим программам необходимо хранить несколько значений, они должны использовать специальную структуру данных, называемую **массивом**. Для объявления массива необходимо указать имя, тип массива и количество значений, которые массив будет хранить.

Массив представляет собой переменную, способную хранить одно или несколько значений. Подобно обычным переменным, массив должен иметь тип (например, `int`, `char` или `float`) и уникальное имя. В дополнение к этому следует указать количество значений, которые массив будет хранить. Все сохраняемые в массиве значения должны быть одного и того же типа. Другими словами, программа не может поместить значения типа `float`, `char` и `long` в один и тот же массив. Следующее объявление создает массив с именем `test_scores`, который может вмещать 100 целых значений для тестовых очков :

```
int test_scores[100];
```

Когда компилятор C++ встречает объявление этой переменной, он распределит достаточно памяти для хранения 100 значений типа `int`. Значения, хранящиеся в массиве, называются элементами массива.

Для обращения к определенным значениям, хранящимся в массиве, используется значение индекса, которое указывает на требуемый элемент. Например, для обращения к первому элементу массива `test_scores` следует использовать значение индекса 0. Для обращения ко второму элементу, индекс 1. Подобно этому, для обращения к третьему элементу, индекс 2. Как показано Указатели. Ссылки, первый элемент массива всегда имеет индекс 0, а значение индекса последнего элемента на единицу меньше размера массива: Важно помнить, что C++ всегда использует 0 для индекса первого элемента массива, а индекс крайнего элемента на единицу меньше размера массива.

Если программа использует массив, обычной операцией является использование индексной переменной для обращения к элементам массива. Например, предположим, что переменная `i` содержит значение 3, следующий оператор присваивает значение 400 элементу `test_scores[3]`: `test_scores[i] = 400;`

C++ позволяет программам инициализировать переменные при объявлении. То же верно и для массивов. При объявлении массива вы можете указать первоначальные

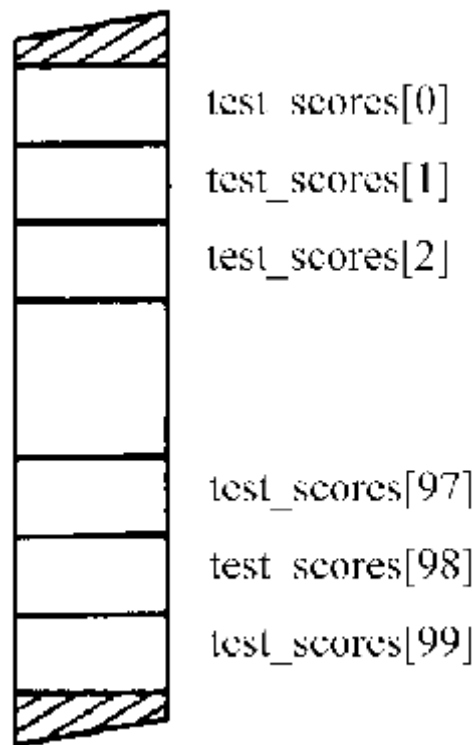


Рис. 3.1: Индексирование элементов массива в C++

значения, поместив их между левой и правой фигурными скобками, следующими за знаком равенства. Например, следующий оператор инициализирует массив `values`:

```
int values[5] = { 100, 200, 300, 400, 500};
```

Подобным образом следующее объявление инициализирует массив с плавающей точкой:

```
float salaries[3] = { 25000.00, 35000.00, 50000.00};
```

Если вы не указываете первоначальное значение для какого-либо элемента массива, большинство компиляторов C++ будут инициализировать такой элемент нулем. Например, следующее объявление инициализирует первые три из пяти элементов массива

```
int values[5] = { 100, 200, 300};
```

Программа не инициализирует элементы `values[3]` и `values[4]`. В зависимости от вашего компилятора, эти элементы могут содержать значение 0<sup>5</sup>.

Если вы не указываете размер массива, который вы инициализируете при объявлении, C++ распределит достаточно памяти, чтобы вместить все определяемые элементы. Например, следующее объявление создает массив, способный хранить четыре целочисленных значения `int numbers[] = { 1, 2, 3, 4};`. Более детально массивы будут рассмотрены далее Указатели. Ссылки.

### Строки

Функции для манипулирования C-строками объявлены в заголовочном файле `string.h` (или по стандарту 1998 г. в `cstring`). Наиболее часто используются

<sup>5</sup> Но лучше на это не уповать.

функции `strlen`, `strcpy`, `strcmp`, `strcat`.

Функция `strcpy` имеет следующий прототип:

```
char *strcpy(char *dst, const char *src);
```

Она копирует строку `src` в строку `dst`, включая нулевой символ, и возвращает указатель на строку `dst`. При этом выход за границы массива `dst` не контролируется:

```
char s1[6] = "Hello", s2[20] = "Good bye";
strcpy(s2, s1);
strcpy(s1, s2); // Runtime error
```

Для определения длины строки служит функция `strlen` с прототипом

```
size_t *strlen(const char *src);
```

Использовать ее предельно просто:

```
char s3[20] = "abracadabra";
size_t sz = strlen(s3); // sz==11
```

Очевидно, что для вычисления длины строки функция `strlen` должна просканировать строку до конца. Возможная реализация `strlen` выглядит так:

```
size_t *strlen(const char *p)
{
    size_t len = 0;
    while ( *p++ )
        len++;
    return len;
}
```

Для добавления одной строки к другой используется функция `strcat` с прототипом

```
char *strcat(char *dst, const char *src);
```

Данная функция добавляет содержимое строки `src` в конец строки `dst` и возвращает полученную строку. Например:

```
char s[20], s1[] = "love";
strcpy(s, "I ");
strcat(s, s1);
strcat(s, " C++"); // s == "I love C++"
```

Возможная реализация `strcat` приведена ниже:

```
char *strcat(char *dst, const char *src)
{
    while( *dst )
        p++;
    while ( *dst++ = *src++ );
    return dst;
}
```

Отметим, что если длина исходной строки известна заранее, то вместо `strcat` можно воспользоваться `strcpy`:

```
strcpy(s, "I ");
strcpy(s + 2, s1);
strcpy(s + 7, " C++");
```

Такой алгоритм эффективнее, так как строка, к которой происходит добавление, не сканируется в поисках завершающего нуля. Функция `strcmp` предназначена для лексикографического сравнения строк. Она имеет прототип

```
int strcmp(const char *p, const char *q);
```

и возвращает 0 если строки совпадают, положительное число, если первая строка больше второй, и отрицательное – если вторая больше первой. Более точно: производится посимвольное сравнение строк до обнаружения пары различных символов или до конца одной из строк и возвращается разность кодов первых не совпавших символов или 0.

Возможная реализация `strcmp` имеет вид:

```
int strcmp(const char *p, const char *q)
{
    while( *p == *q && *p ) {
        p++;
        q++;
    }
    return *p - *q;
}
```

В заключение данного пункта приведем менее употребительные функции работы с С-строками<sup>6</sup>:

`char *strncpy(char *p, const char *q, int n);` – то же, что и `strcpy`, но копируется максимум `n` символов.

`char *strncat(char *p, const char *q, int n);` – то же, что и `strcat`, но добавляется максимум `n` символов.

`int strncmp(const char *p, const char *q, int n);` – то же, что и `strcmp`, но сравнивается максимум `n` символов.

`char *strchr(const char *p, char c);` – возвращает адрес первого вхождения символа `c` в строку `p` (или 0, если символ не найден).

`char *strrchr(const char *p, char c);` – возвращает адрес последнего вхождения символа `c` в строку `p` (или 0, если символ не найден).

`char *strstr(const char *p, const char *q);` – возвращает адрес первого вхождения подстроки `q` в строку `p` (или 0, если подстрока не найдена).

`char *strpbrk(const char *p, const char *q);` – возвращает адрес первого вхождения в строку `p` какого-либо символа из строки `q` (или 0, если совпадений не обнаружено).

`size_t strspn(const char *p, const char *q);` – возвращает число начальных символов в строке `p`, которые не совпадают ни с одним из символов из строки `q`.

`size_t strcspn(const char *p, const char *q);` – возвращает число начальных символов в строке `p`, которые совпадают с одним из символов из строки `q`.

`char *strtok(char *p, const char *q);` – последовательность вызовов функции `strtok` разбивает строку `p` на лексемы, разделенные символами из строки `q`. При первом вызове в качестве `p` передается указатель на строку, которую надо разбить на лексемы, при всех последующих – нулевой указатель. При этом значение указателя, с которого должен начинаться поиск следующей лексемы, сохраняется

<sup>6</sup> На самом деле, из-за соображений безопасности лучше использовать приведенные функции, т.к. в них упор идет не на окончание строки (NULL), а на передаваемую длину строки. Поэтому испорченная строка (без окончания NULL) не сможет поломать программу.

в некоторой системной переменной. Функция `strtok` возвращает указатель на найденную лексему или 0, если лексем больше нет. Она также модифицирует исходную строку, вставляя нулевой символ после найденной лексемы.

### 3.4 Составные типы данных

**Структуры** В большинстве случаев программам необходимо группировать связанную информацию разного типа. Например, программа работает с информацией о служащих. Она должна отслеживать данные о фамилии, возрасте, окладе, адресе, номере служащего и т. д. Для хранения этой информации программе потребуются переменные типа `char`, `int`, `float`, а также символьные строки.

Структура определяет шаблон, с помощью которого программа может позднее объявить одну или несколько переменных. Другими словами, программа сначала определяет структуру, а затем объявляет переменные типа этой структуры. Для определения структуры программы используют ключевое слово `struct` (POD Типы объектов в C++. POD, non-POD), за которым обычно следует имя и левая фигурная скобка. Следом за открывающей фигурной скобкой вы указываете тип и имя одного или нескольких элементов. За последним элементом вы размещаете правую закрывающую фигурную скобку. В этот момент вы можете (необязательно) объявить переменные данной структуры. Определение, содержащее информацию о служащем:

Листинг 3.3: Определение, содержащее информацию о служащем

```
struct employee {  
    char name [64];  
    long employee_id;  
    float salary;  
    char phone[10];  
    int id_number;  
};
```

В данном случае определение не объявляет какие-либо переменные типа этой структуры. После того как определена структура, можно объявить переменные типа этой структуры, используя имя структуры (иногда называемое **структурным тэгом**), как показано :

```
employee boss, worker;  
/** And C style */  
struct employee boss, worker;
```

Структура позволяет группировать информацию, называемую элементами, в одной переменной. Чтобы присвоить значение элементу или обратиться к значению элемента, используется оператор C++ точку (`.`). Например, следующие операторы присваивают значения различным элементам переменной с именем `worker` типа `employee`:

```
worker.employee_id = 12345;  
worker.salary = 25000.00;  
worker.id_number = 102;
```

Для обращения к элементу структуры указывается имя переменной, за которым следует точка и имя элемента (поля).

#### Объединения

По мере усложнения программ могут потребоваться разные способы просмотра части информации. Кроме того, программе может потребоваться работать с двумя или несколькими значениями, используя при этом только одно значение в каждый момент времени. В таких случаях для хранения данных ваши программы могут использовать объединения.

Внутри программ объединения C++ очень похожи на структуры. Например, следующая структура определяет объединение с именем `distance`, содержащее два элемента:

```
union distance {
    int miles;
    long meters;
};
```

Как и в случае со структурой, описание объединения не распределяет память. Вместо этого описание предоставляет шаблон для будущего объявления переменных. Чтобы объявить переменную объединения, вы можете использовать любой из следующих форматов:

```
union distance {
    int miles;
    long meters;
} moscow;
/** or */
union distance {
    int miles;
    long meters;
};
distance moscow;
```

Как видите, данное объединение содержит два элемента: `miles` и `meters`. Эти объявления создают переменные, которые позволяют вам хранить расстояния до указанных стран. Как и для структуры, ваша программа может присвоить значение любому элементу. Однако в отличие от структуры значение может быть присвоено только одному элементу в каждый момент времени. Когда вы объявляете объединение, компилятор C++ распределяет память для хранения самого большого элемента объединения. В случае объединения `distance` компилятор распределяет достаточно памяти для хранения значения типа `long`, как показано на рисунке Составные типы данных. Предположим, что программа присваивает значение элементу `miles`:

```
moscow.miles = 12123;
```

Если далее в программе идет присваивание значения элементу `meters`, значение, присвоенное элементу `miles`, теряется.

Также существует такое понятие как **анонимное объединение**. Анонимное объединение представляет собой объединение, у которого нет имени. C++ предоставляет анонимные объединения, чтобы упростить использование элементов объединений, предназначенных для экономии памяти или создания псевдонимов для определенного значения. Например, предположим, что программе требуются две переменные `miles` и `meters`. Кроме того, предположим, что программа использует только одну из них каждый данный момент времени. В этом случае программа могла бы использовать элементы объединения, подобного уже обсуждавшемуся объединению `distance`, а именно `name.miles` и `name.meters`. Следующий оператор создает анонимное (безымянное) объединение:



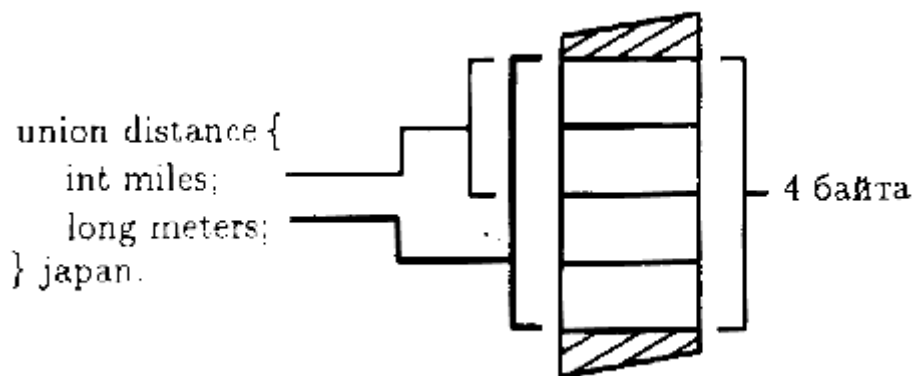


Рис. 3.2: C++ распределяет память, достаточную для хранения только самого большого элемента объединения

```
union {
    int miles;
    long meters;
};
```

Как видите, объявление не использует имя объединения и не объявляет переменную объединения. Программа, в свою очередь, может обращаться к элементам с именами `miles` и `meters`.

В совокупности с битовыми масками, объединения являются мощнейшим инструментом.

### 3.5 Указатели. Ссылки.

Данный раздел, переработанный и исправленный, был взят из методического указания для студентов механико-математического факультета **ЮФУ** [?]. За что автору огромное спасибо.

Указатели (`pointers`) и ссылки (`references`) в том или ином виде присутствуют в большинстве языков программирования. При этом значение слова «ссылка» может иметь несколько различных смыслов. Чтобы лучше понять значения этих слов, приведем несколько примеров из повседневной жизни. Когда мы произносим имя человека, то ссылаемся на определенного человека. Можно сказать, что имя человека является ссылкой на него. Ссылкой на этого же человека является и фраза «владелец дома по адресу ...». Ссылкой на дом является адрес этого дома, его название (например, «Дом книги») или его однозначная характеристика (например, «красный дом в конце квартала»). В роли указателя может выступать табличка с именем объекта и инструкцией по его нахождению. Дадим более строгие определения. Будем называть ссылкой (в широком смысле) некоторое имя или фразу, однозначно идентифицирующие объект. Очевидно, несколько ссылок могут обозначать один и тот же объект, представляя его различные имена. Указателем (в широком смысле) назовем объект, хранящий информацию о местонахождении другого объекта. По существу, указатель представляет собой объект, хранящий ссылку на другой объект. Указатель в процессе своего существования может изменить значение содержащейся в нем ссылки, указывая тем самым на другой объект.

В языках программирования в роли ссылки чаще всего выступает имя объекта или адрес этого объекта. Указатель же – это объект, хранящий адрес другого объекта. Подчеркнем, что, в отличие от терминологии, принятой в объектно-ориентированных языках программирования, везде под объектом понимается область памяти, имеющая определенный тип. Очень близким по смыслу является понятие переменной – области памяти, имеющей тип и имя. Таким образом, согласно нашей терминологии, переменная – это объект, имеющий имя.

Указатель в C++ – это переменная, хранящая адрес некоторого объекта. Если объект имеет тип *T*, то указатель на него описывается следующим образом *T\* p*; . При объявлении нескольких указателей символ *\** обязателен перед именем каждой переменной. Так, в объявлении `double *p1, p2, *p3`; переменная *p2* принадлежит к типу `double`. Записи *T\* p* и *T \*p* равноценны. Запись *T\* p* обычно читается как «*p* принадлежит к типу указатель на *T*», а запись *T \*p* – как «*p* является указателем на объект типа *T*».

Указателю можно присвоить либо значение 0<sup>7</sup>, либо адрес объекта, используя унарный оператор взятия адреса `&`. Например:

```
int i, a[10];
int* pi = &i, *pa, *pb = 0;
pa = &a[9];
```

Здесь указатель *pi* инициализируется при объявлении адресом переменной *i*, указатель *pb* – нулем, а указатель *pa* инициализируется присваиванием. Присваивание указателю нулевого значения означает, что он не указывает ни на один объект. Массив указателей и указатель на массив объявляются следующим образом:

```
int* b[10];           // array pointers
int (*paa)[10] = &a; // Pointer to array
```

Можно также объявить указатель на указатель:

```
char c;
char* pc = &c;
char** ppc = &pc;
```

Указатели, инициализированные значением 0, называются нулевыми и ни на что не указывают. Бывают случаи, когда нас интересует просто значение адреса, а не тип указываемого объекта. С этой целью в язык C++ введены указатели `void *`, способные хранить адрес любого объекта. В отличие от типизированных указателей, рассмотренных выше, указатели `void *` называются нетипизированными.

Для доступа к объекту через указатель используется оператор разыменования `*`, осуществляющий так называемую косвенную адресацию: `*p` означает «объект, на который указывает *p*». По существу, `*p` представляет собой ссылку (в широком смысле) на объект. Например, если действуют описания предыдущего пункта, то в результате выполнения следующего фрагмента:

```
*pi = 5;
(*paa)[9] = 77;
**ppc = 'a';
```

<sup>7</sup> Для большей наглядности лучше использовать макроопределение `NULL`, которое определено лишь для того, чтобы явно показать нулевой указатель. Но надо быть осторожными т.к. не во всех реализациях присутствует `NULL`.

переменной `pi` будет присвоено значение 5, переменной `раа[9]` – значение 77, а переменной `ppc` – значение 'а'. Отметим, что во второй строке скобки обязательны, так как оператор `[]` имеет более высокий приоритет, чем оператор `*`.

Если `p` является указателем на структуру `s` с полем `a`, то оператор `->` в записи `p->a` используется для сокращения записи `(*p).a`. Например:

```
struct point {
    double x, y;
};
point t, *pt = &t;
pt->x = pt->y = 1;
```

Попытка разыменования нулевого указателя приводит к ошибке при выполнении программы, попытка же разыменования указателя `void *` вызовет ошибку на этапе компиляции.

Одной из распространенных ошибок является разыменование неинициализированных указателей. Например, в любом из следующих случаев результат работы программы непредсказуем:

```
char* s;
*s = 'a'; // Exception
std::cin >> s; // Exception
```

Отметим, что в последнем случае, где выводится строка, на которую указывает `s`, разыменование присутствует неявно и происходит внутри оператора ввода из потока.

Указатель на один тип нельзя присвоить указателю на другой тип без явного преобразования типов. Исключение составляет указатель `void *`, который трактуется как указатель на некоторый участок памяти. Он называется родовым указателем и может получать в качестве значения указатель на любой другой тип без явного преобразования. Например:

```
int i;
void* pi = &i;
```

Напомним, что указатель `void *` нельзя разыменовывать. Каким же образом получить доступ к переменной `i` через указатель `pi`? Для этого необходимо использовать оператор приведения типа `static_cast`. Он преобразует объект к типу указателя, записанному в угловых скобках `int* pi1 = static_cast<int*>pi;`.

В некоторых старых компиляторах оператор `static_cast` отсутствует, поэтому приходится использовать приведение типа в старом стиле `int* pi1 = (int*)pi;`.

Отметим, что явные преобразования типов в большинстве случаев являются потенциально опасными и должны применяться с крайней осторожностью. Так, в следующем примере `double d = *static_cast<double*>pi;` содержимое переменной `d` непредсказуемо. Именно потенциальная опасность операторов приведения типа привела к необходимости систематизации ситуаций, в которых используется приведение. В новой редакции C++ старый способ приведения заменен на четыре новых, различающихся по степени безопасности. Среди них уже рассмотренный нами оператор `static_cast`, оператор снятия константности `const_cast` (он будет рассмотрен далее), а также операторы `reinterpret_cast` (для преобразования принципиально различных типов) и `dynamic_cast` (для преобразования полиморфных типов, связанных иерархией наследования).

Свойство указателей `void *` хранить данные разнородных типов используется при создании так называемых родовых (`generic`) массивов, т. е. массивов, хранящих разнотипные объекты. Например:

```
#include <iostream>

int i = 5;
char c = 'u';
double d[2] = {3, 6};
void* generic[] = { &i, &c, &d };
std::cout << *static_cast<int *>generic[0] << ' '
          << *static_cast<char *>generic[1] << ' '
          << (static_cast<double *>generic[2])[1];
```

В некоторых немногочисленных случаях необходимо уметь модифицировать значение константного объекта. Например:

```
int i = 3;
const int* pi = &i;
int* pi = pi; // Compile error
```

Несмотря на то, что мы заведомо знаем, что `pi` указывает на неконстантный объект, изменить значение этого объекта мы не можем. В этом случае требуется явное приведение типа с помощью оператора приведения типа `const_cast`, «снимающего» константность `int* pi = const_cast<int *>(pi);`. Теперь данные, адрес которых хранится в переменной `pi`, можно косвенно изменить через указатель `pi *pi=4;`. То же самое можно сделать, используя `const_cast` в левой части оператора присваивания `*const_cast<int *>(pi)= 4;`. Вновь подчеркнем потенциальную опасность операторов явного приведения типа. В следующем примере:

```
const int n = 5;
int* pi = const_cast<int *>(&n);
*pi = 7;
```

изменяется значение настоящей константы, что недопустимо!

Над указателями можно совершать ряд арифметических действий. При этом предполагается, что если указатель `p` относится к типу `T *`, то `p` указывает на элемент некоторого массива типа `T`. Тогда `p + 1` является указателем на следующий элемент этого массива, а `p - 1` – указателем на предыдущий элемент. Аналогично определяются выражения `p + n`, `n + p` и `p - n`, а также действия `p++`, `p--`, `++p`, `--p`, `p += n`, `p -= n`, где `n` – целое число. Важно отметить, что арифметические действия с указателями выполняются в единицах того типа, к которому относится указатель. То есть `p + n`, преобразованное к целому типу, содержит на `sizeof(T) * n` большее значение, чем `p`.

Из равенства `p + n == p1` следует, что `p1 - p == n`. Именно так вводится оператор разности двух указателей: его значением является целое, равное количеству элементов массива от `p` до `p1`. Отметим, что это – единственный случай в языке, когда результат бинарного оператора с операндами одного типа принадлежит к принципиально другому типу. Сумма двух указателей не имеет смысла и поэтому не определена. Не определены также арифметические действия над нетипизированными указателями `void *`. Наконец, все указатели, в том числе и нетипизированные, можно сравнивать, используя операторы отношения `>`, `<`, `>=`, `<=`, `==`, `!=`.

Поясним сказанное примерами.

Листинг 3.4: Сумма элементов массива

```
int a[10], s;
for (int* p = &a[0]; p <= &a[9]; p++)
    s += *p;
```

Листинг 3.5: Ивертирование массива

```
for (int* p = &a[0], *q = &a[9]; p < q; p++, q--)
    swap(p, q);
```

Подробнее про функцию `swap` будет рассказано далее [Указатели. Ссылки](#).

Указатели и массивы тесно взаимосвязаны. Имя массива может быть неявно преобразовано к константному указателю на первый элемент этого массива. Так, `&a[0]` равноценно `a`. Вообще, верна формула `a[n] == a + n`; то есть адрес  $n$ -того элемента массива есть увеличенный на  $n$  элементов указатель на начало массива. Разыменовывая левую и правую части, получаем основную формулу, связывающую массивы и указатели:

$$a[n] == *(a + n);$$

Данная формула, несмотря на простоту, требует нескольких пояснений. Во-первых, компилятор любую запись вида `a[n]` интерпретирует как `*(a + n)`. Во-вторых, формула поясняет, почему в C++ массивы индексируются с нуля и почему нет контроля выхода за границы диапазона. Наконец, используя эту формулу, мы можем записать следующую цепочку равенств: `a[n] == *(a + n) == *(n + a) == n[a]`; Таким образом, элемент массива `a` с индексом 2 можно обозначить не только как `a[2]`, но и как `2[a]`.

Из связи массивов и указателей вытекает способ передачи массивов в функции – с помощью указателя на первый элемент. Более того, следующие прототипы функций Функции полностью эквивалентны:

```
void print(int a[10], size_t n);
void print(int a[], size_t n);
void print(int *a, size_t n);
```

В частности, нетрудно проверить, что `sizeof(a)` внутри функции `print()` во всех трех случаях совпадает с `sizeof(int*)`. Таким образом, внутри функции теряется информация о размере массива, поэтому размер необходимо передавать явно как еще один параметр. Заметим также, что массив в языке C++ нельзя передать по значению (как в языке Паскаль): изменение элемента массива внутри функции `print()` всегда приводит к изменению фактического параметра-массива. Это же замечание справедливо для строк `char *`, являющихся указателями на символьные массивы. Подчеркнем еще раз, что указатель, к которому преобразуется имя массива, – константный. В частности, это означает, что имени массива нельзя присвоить значение:

```
int a[10], b[10];
a = b; // Compile error
```

Многомерные массивы в C++ конструируются из одномерных. Рассмотрим их создание и использование на примере двумерных массивов. `int a[3][4]`; вводит массив из трех элементов, каждый из которых имеет тип `int[4]`, т. е. представляет собой одномерный массив из четырех целых. Если первый индекс двумерного массива трактовать как номер строки матрицы, то можно сказать, что двумерный массив

хранится в оперативной памяти построчно: вначале первая строка `a[0]`, затем вторая – `a[1]` и затем третья – `a[2]` Двумерный массив: распределение памяти.

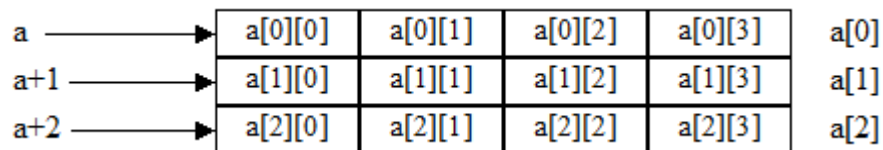


Рис. 3.3: Двумерный массив: распределение памяти

Расшифруем выражение `a[1][2]`, воспользовавшись формулой  
`a[1][2] == *( *( a + 1) + 2 );`.

Имя массива `a` можно трактовать как указатель на начало одномерного массива с элементами типа `int[4]`, т. е. на его первый элемент `a[0]`. Поскольку арифметика указателей чувствительна к типу, то запись `a + 1` трактуется как указатель на второй элемент одномерного массива типа `int[4]`, т. е. как указатель на `a[1]`. Разыменовывая указатель `a + 1`, получаем `*(a + 1)`, или `a[1]`, представляющий собой одномерный массив элементов типа `int`. Имя этого массива `a[1]` преобразуется к указателю на его первый элемент, т. е. на элемент `a[1][0]`. Если к этому указателю на тип `int` прибавить 2, то `a[1] + 2` будет указывать на элемент `a[1][2]`. После разыменования указателя `a[1] + 2` мы и получим элемент `a[1][2]`

`*( *( a + 1) + 2) == *( a[1] + 2) == a[1][2];`.

### Ссылки

Ссылка в C++ – это альтернативное имя (псевдоним) объекта. Это определение уже общего определения ссылки, данного в начале (напомним, что ссылка в широком смысле – это любое имя объекта). Запись `T&` обозначает ссылку на объект типа `T`. Например, в следующем фрагменте:

```
int i = 1;
int& r = i;
```

объявляется ссылка на переменную `i`, в результате чего как `r`, так и `i` ссылаются на один и тот же объект целого типа. Теперь значение переменной `i` можно изменить через ссылку `r`:

```
r = 2; // i value 2
r++; // i value 3
```

В момент объявления ссылка обязательно должна инициализироваться объектом соответствующего типа. После объявления повторная инициализация ссылки невозможна. Обратите внимание, что для обозначения ссылочного типа используется тот же символ `&`, что и для оператора взятия адреса. Что именно имеется в виду, ясно из контекста: знак `&` после имени типа используется для объявления ссылки, в остальных случаях он обозначает оператор взятия адреса. Так, в результате выполнения оператора `std::cout << &r`; выводится адрес объекта, представляемого ссылкой `r`. В ситуациях, когда основное имя объекта является составным, ссылка позволяет его сократить:

```
int a[3][4];
int& last = a[2][3];
```

Заметим также, что могут существовать объекты, вообще не имеющие собственного имени; ссылка для них выступает в роли единственного имени и предоставляет единственный способ доступа:

```
int& rd = *new int;
rd = 5;
delete &rd;
```

Более детальную информацию о динамическом выделении памяти будет рассказано далее Работа с памятью. Ссылку можно представлять себе как константный указатель, который всегда разыменован. Однако необходимо помнить, что, в отличие от указателя, память под ссылку не выделяется, т. е. сама ссылка не является объектом! Именно поэтому не имеет смысла объявление указателя на ссылку, ссылки на ссылку или массива ссылок:

```
int&& r2 = r; // Compile error
int&* r3 = r; // Compile error
int& a[3]; // Compile error
```

Ссылка же на указатель является распространенной практикой:

```
int *p;
int*& rp = p;
rp++;
```

Как и в случае указателей, `const T&` означает ссылку на константу. В отличие от обычной ссылки `T&`, ее можно инициализировать константным объектом:

```
int i = 1;
const int c = 5;
int& rv = c; // Compile error
int& rv1 = 5; // Compile error
const int& rc = c; // Passed
const int& rc1 = 5; // Passed
const int& rc2 = i; // Passed
```

Отметим, что некоторые компиляторы рассматривают третью и четвертую строки лишь потенциально ошибочными, выдавая вместо сообщения об ошибке предупреждающее сообщение. Константная ссылка `int& const` смысла не имеет. Для снятия константности связанных со ссылкой данных следует использовать оператор `const_cast`:

```
int i = 1;
const int& ri = i;
int& ri1 = const_cast<int &>(ri);
```

Ссылки могут использоваться в качестве параметров функции, которая изменяет значение передаваемого ей объекта. Рассмотрим реализацию функции `swap`, с использованием ссылок:

```
void swap(int& a, int& b)
{
    int t = a;
    a = b;
    b = t;
}
```



```
/** ... */
int i = 3, j = 4;
swap(i, j);
```

При вызове `swap(i, j)` ссылки `a` и `b` внутри функции становятся новыми именами для переменных `i` и `j`, в результате чего работа происходит именно с самими переменными `i` и `j`, а не с их копиями. Отметим также, что попытка вызвать функцию `swap` с параметрами другого типа неизбежно приведет к ошибке:

```
unsigned int u1 = 5, u2 = 6;
swap(u1, u2); // Exception
```

Внутренний механизм передачи параметров-ссылок тот же, что и для указателей: в программный стек копируется не значение передаваемого объекта, а указатель на него, и ссылка внутри функции выполняет роль этого указателя, который всегда разыменован. В следующем примере в качестве параметра используется ссылка на указатель:

```
void next_space(char*& p)
{
    while ( *p != ' ' && *p )
        p++;
}
```

Как следует из названия, функция сканирует строку, на которую указывает `p`, в поисках пробела. В результате своей работы она возвращает в `p` либо адрес первого пробела, либо адрес конца строки. Ссылки на константы широко используются для передачи больших параметров, чтобы избежать копирования значений этих параметров в стек. Например, в функции

```
double det(const Matrix& A) {
    /** */
}
```

параметр `A` передается как ссылка на константу, что подчеркивает, с одной стороны, его неизменность внутри функции, а с другой – то, что он имеет большой размер и поэтому передается как указатель. Отметим, что ссылка на константу позволяет безболезненно указывать в качестве фактического параметра объект другого типа: при вызове совершается неявное преобразование к типу формального параметра, соответствующее значение записывается во временную переменную, после чего ссылка связывается с этой безымянной временной переменной. Например, в ситуации

```
void f(const double& d);
/** ... */
f(3);
```

будет создана временная переменная типа `double`, в нее будет записано значение `3` (принадлежащее изначально к типу `int`), после чего ссылка `d` внутри функции будет являться псевдонимом этой временной переменной. Напомним, что такая временная переменная будет существовать все время жизни ссылки, т. е. до выхода из функции `f`.

В программировании нередко возникают ситуации, когда функции необходимо передавать в качестве параметров других функций. С этой целью в языке C++ используются



ссылки и указатели на функции. Указатели на функции могут использоваться также для изменения какого-либо действия в ходе выполнения программы.

Начнем с примера. Пусть имеется функция

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

Указатель на такую функцию объявляется следующим образом:

```
double (*pf)(double a, double b);
```

Затем он может быть инициализирован адресом этой функции, после чего функция add может быть вызвана косвенно через указатель:

```
pf = &add;
std::cout << (*pf)(2, 3);
```

Разыменование указателя на функцию при помощи оператора \* не является обязательным. Также не является обязательным использование оператора & для получения адреса функции. Таким образом, предыдущий код можно упростить:

```
pf = add;
std::cout << pf(2, 3);
```

Для повторного использования типа указателя на функцию удобно присвоить ему имя при помощи директивы typedef:

```
typedef double (*FUN)(double a, double b);
```

После этого указатель на функцию можно описать следующим образом:

```
FUN pf = add;
```

Указатели на функции имеют все преимущества обычных указателей. В процессе работы указатель может менять свое значение, обеспечивая тем самым различное поведение программы в зависимости от действий пользователя. Можно также объявить массив указателей на функции:

```
double add(double a, double b);
double sub(double a, double b);
double mul(double a, double b);
double div(double a, double b);

FUN f[4] = { add, sub, mul, div };
int n;
double a, b;
std::cin >> n >> a >> b;
std::cout << f[n](a, b);
```

Ссылка на функцию объявляется аналогично:

```
double (&rf)(double a, double b) = add;
```

Однако, поскольку ссылка, в отличие от указателя, не может в процессе работы поменять функцию, с которой связана, диапазон применения ссылок на функции

ограничивается передачей параметров-функций. Разумеется, указатель или ссылка на функцию может инициализироваться только функциями того же типа, т. е. имеющими те же количество и типы формальных параметров, а также тип возвращаемого значения, что и тип указателя (ссылки).

**Замечание.** Так как `inline` – функции не являются функциями в собственном смысле этого слова и не имеют адреса, использовать указатели или ссылки на `inline` – функции запрещено.

Для иллюстрации применения указателей и ссылок на функции рассмотрим два примера.

**Пример 1.** Применение функции, передаваемой в качестве параметра, ко всем элементам диапазона (о понятии диапазона Указатели. Ссылки):

```
typedef void (&FUN)( double& );

void for_each(double* a, double* b, FUN f)
{
    while( a != b )
        f( *a++ );
}
```

**Пример 2.** Сортировка массива с элементами произвольного типа и критерием сравнения, передаваемым в качестве параметра. В данном примере реализована функция `ssort`, сортирующая методом пузырька массив данных произвольного типа. Количество элементов задается параметром `n`, размер каждого элемента – параметром `sz`, функция сравнения – параметром `cmp`. Поскольку типы элементов заранее неизвестны, указатель на первый элемент передается как `void *`. Внутри функции `ssort` он преобразуется к типу `char *` для возможности работы с адресной арифметикой. Функция `memswap` меняет местами две области памяти размера `sz`. В функцию сравнения передаются два указателя `void *` на сравниваемые элементы массива. Каждая конкретная функция сравнения вначале преобразует эти указатели к нужному типу и затем осуществляет собственно сравнение элементов. В программе, демонстрирующей варианты применения функция `ssort`, реализована сортировка целых чисел (по возрастанию и убыванию), текстовых строк, а также структур (по нескольким полям). Далее приводится полный текст программы и результаты вывода.

```
#include <iostream>

typedef int (*CMP)(const void *,const void *);

inline void swap(char& a, char& b)
{
    char temp = a;
    a = b;
    b = temp;
}

void memswap(char* a, char* b, size_t sz)
{
    for (int k = 0; k < sz; k++)
        swap( *a++, *b++);
}
```

```
void ssort(void* base, size_t n, size_t sz, CMP cmp)
{
    for (int i = 1; i < n; i++)
        for (int j = n - 1; j >= i; j--) {
            char* bj = (char *)base + j * sz;
            if ( cmp(bj, bj - sz) )
                memswap(bj, bj - sz, sz);
        }
}

struct database {
    char *name;
    int age;
};

int less_int(const void* p, const void* q)
{
    return *(int*)p < *(int*)q;
}

int greater_int(const void* p, const void* q)
{
    return *(int *)p > *(int *)q;
}

int less_str(const void * p, const void * q)
{
    return strcmp( *(char **)p, *(char **)q) < 0;
}

int less_age(const void * p, const void * q)
{
    return ( (database *)p)->age < ((database *)q)->age;
}

int less_name(const void * p, const void * q)
{
    return strcmp( ((database *)p)->name, ((database *)q)->name) < 0;
}

void print (int* mas, int n)
{
    for (int i = 0; i < n; i++)
        std::cout << mas[i] << " ";
    std::cout << std::endl;
}

void print (char** mas, int n)
{
    for (int i = 0; i < n; i++)
        std::cout << mas[i] << " ";
    std::cout << std::endl;
}
```

```
}

void print (database* mas, int n)
{
    for (int i = 0; i < n; i++)
        std::cout << mas[i].name << " " << mas[i].age << std::endl;
}

const int n = 10, m = 5;
int mas[n] = { 1, 5, 2, 6, 3, 7, 12, -1, 6, -3 };
char* strmas[n] = {
    "adg",
    "dfgj",
    "jk",
    "asg",
    "gjh",
    "sdh",
    "hj",
    "sd",
    "kfj",
    "sdadgh"
};
database d[m] = {
    { "Petrov", 32 },
    { "Ivanov", 24 },
    { "Kozlov", 21 },
    { "Oslov", 20 },
    { "Popov", 18 }
};

void main()
{
    ssort(mas, n, sizeof(int), less_int);
    print(mas, n);
    ssort(mas, n, sizeof(int), greater_int);
    print(mas, n);
    ssort(strmas, n, sizeof(char *), less_str);
    print(strmas, n);
    std::cout << std::endl;
    ssort(d, m, sizeof(database), less_age);
    print(d, m);
    std::cout << std::endl;
    ssort(d, m, sizeof(database), less_name);
    print(d, m);
}
```

### 3.6 Работа с памятью

В распоряжении пользователя имеется несколько разных способов работы с памятью. Также переменные программы могут располагаться в разных областях и

для хранения данных могут использовать разные типы хранения значений. Всего имеется около 3 типов хранения значений переменных:

- \* стек Организация памяти;
- \* куча Организация памяти;
- \* непосредственно тело исполняемого модуля (в одной из секций) Организация исполняемого модуля.

Стек – структура данных, в которой доступ к элементам осуществляется по типу LIFO. Стек в архитектурах x86 растет вниз, т.е. самые крайние элементы имеют меньший адрес. Все переменные определенные в функциях или процедурах программы,



Рис. 3.4: Структура и принцип действия стека

помещаются в стек. Следует быть осторожными, т.к. можно с легкостью его переполнить. Пример кода, который переполняет стек и получить не тот результат Работа с памятью, который требуется.

Листинг 3.6: Пример кода, который переполняет стек

```
int main()
{
    int nStack[100000000];
    return 0;
}
```



Рис. 3.5: Результат переполнения стека

Как можно понять, стек ограниченный ресурс. Нужно быть предельно внимательным при его использовании.

Куча (Heap) – структура данных при помощи которой реализована динамически распределяемая память. Динамическая память выделяется при помощи стандартных средств языка (операторов) или вызовов функций стандартной библиотеки. Для C++ существуют несколько таких вызовов, их можно разделить на две группы: выделение и освобождение памяти.

```
void* operator new (std::size_t size) throw (std::bad_alloc);
void* operator new (std::size_t size,
    const std::nothrow_t& nothrow_constant) throw();
void* operator new (std::size_t size, void* ptr) throw();

void* operator new[] (std::size_t size) throw (std::bad_alloc);
void* operator new[] (std::size_t size,
    const std::nothrow_t& nothrow_constant) throw();
void* operator new[] (std::size_t size, void* ptr) throw();
```

```

void operator delete (void* ptr) throw ();
void operator delete (void* ptr,
    const std::nothrow_t& nothrow_constant) throw();
void operator delete (void* ptr, void* voidptr2) throw();

void operator delete[] (void* ptr) throw ();
void operator delete[] (void* ptr,
    const std::nothrow_t& nothrow_constant) throw();
void operator delete[] (void* ptr, void* voidptr2) throw();

```

Синтаксис `new` выглядит следующим образом:

```
p_var = new typename;
```

где `p_var` – ранее объявленный указатель типа `typename`. `typename` может подразумевать собой любой фундаментальный тип данных или объект, определенный пользователем (включая, `enum`, `class` и `struct`). Если `typename` – это тип класса или структуры, то он должен иметь доступный конструктор по умолчанию, который будет вызван для создания объекта.

Для инициализации новой переменной, созданной при помощи `new` нужно использовать следующий синтаксис:

```
p_var = new type(initializer);
```

где `initializer` – первоначальное значение, присвоенное новой переменной, а если `type` – тип класса, то `initializer` – аргумент(ы) конструктора.

`new` может также создавать массив Массивы:

```
p_var = new type [size];
```

В данном случае, `size` указывает размерность (длину) создаваемого одномерного массива. Адрес первого элемента возвращается и помещается в `p_var`, поэтому `p_var[n]` означает значение `n`-ого элемента (считая от нулевой позиции).

Память, выделенная при помощи `new`, должна быть освобождена при помощи `delete`, дабы избежать утечки памяти. Массивы, выделенные (созданные) при помощи `new[]`, должны освобождаться (уничтожаться) при помощи `delete[]`.

```

int *p_scalar = new int(5);
int *p_array = new int[5];

```

Инициализаторы не могут быть указаны для массивов, созданных при помощи `new`. Все элементы массива инициализируются при помощи конструктора по умолчанию для данного типа. Если тип не имеет конструктора по умолчанию, выделенная область памяти не будет проинициализирована.

В компиляторах, придерживающихся стандарта ISO C++, в случае если недостаточно памяти для выделения, то генерируется исключение типа `std::bad_alloc`. Выполнение всего последующего кода прекращается, пока ошибка не будет обработана в блоке `try-catch` или произойдет экстренное завершение программы. Программа не нуждается в проверке значения указателя; если не было сгенерировано исключение, то выделение прошло успешно. Реализованные операции определяются в заголовке `<new>`. В большинстве реализаций C++ оператор `new` также может быть перегружен для определения особого поведения.

Любая динамическая память выделенная при помощи `new` должна освобождаться при помощи оператора `delete`. Существует два варианта: один для массивов, другой – для единичных объектов.

```
int *p_var = new int;  
int *p_array = new int[50];  
  
delete[] p_array;  
delete p_var;
```

Необходимо отметить, что стандарт не требует от компилятора создания диагностического сообщения при некорректном использовании `delete`; он в общем случае не может знать, когда указатель указывает на одиночный элемент, а когда – на массив элементов. Более того, использование не соответствующего освобождения является неопределённым поведением.

Для управления динамической памятью в языке C существует 4 функции:

```
#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` – выделение блока памяти заданного размера.

Функция `calloc` – выделение «чистого» блока памяти. В данном случае имеется в виду что память будет заранее очищена.

Функция `realloc` – перераспределение, уменьшение или увеличение размера выделенного ранее блока памяти.

Функция `free` – освобождение ранее выделенной функциями `malloc`, `calloc`, `realloc`.

Размещение глобальных переменных в памяти.

Листинг 3.7: Пример кода, использующего глобальные переменные

```
int nGlobal[100000000];  
  
int main()  
{  
    return 0;  
}
```

## 3.7 Практическая работа

**Массивы** Считая, что переменная `a` описана как `a[3][4]`, расшифруйте выражения (пример Указатели. Ссылки):

1. `*a;`
2. `**a;`
3. `*a+1;`
4. `(*a)[2];`
5. `*(a[2]);`
6. `*a[2];`

7. `*2[a];`
8. `*1[a + 1];`
9. `&a[0][0];`
10. `2[a][3].`

1. Дан массив целых. Оформить функцию `count_if`, вычисляющую количество элементов в массиве, удовлетворяющих данному условию, передаваемому в качестве параметра (условие должно представлять собой функцию, принимающую параметр типа `int` и возвращающую значение логического типа).
2. Дан массив целых. Заполнить его, передавая в качестве параметра функцию, задающую алгоритм генерации следующего значения, вида `int f()`. Для генерации данная функция может запоминать значения, сгенерированные на предыдущем шаге, либо в глобальных переменных, либо в статических локальных переменных.
3. Дан массив целых, отсортированный по возрастанию. Удалить из него дубликаты.
4. Дан массив целых. Составить функцию `remove_if`, удаляющую из него все элементы, удовлетворяющие условию, передаваемому в качестве параметра.
5. Дан массив чисел и число `a`. Переставить элементы, меньшие `a`, в начало, меняя их местами с предыдущими. Порядок элементов, меньших `a`, а также порядок элементов, больших `a`, не менять.
6. Дан массив целых. Найти в нем пару чисел `a` и `b` с минимальным значением `f(a, b)`, где `f` передается в качестве параметра.
7. Дан массив целых. Составить функцию `accumulate`, применяющую функцию `f(s, a)`, передаваемую в качестве параметра, к каждому элементу `a` массива и записывающую результат в переменную `s`. С ее помощью найти минимальный элемент в массиве, сумму и произведение элементов массива.
8. Дан массив целых. Сформировать по нему массив, содержащий длины всех серий (подряд идущих одинаковых элементов). Одиночные элементы считать сериями длины 1.
9. Дан массив целых. Из каждой серии удалить один элемент.
10. Дан массив целых. Удалить все серии, длина которых меньше `k`.
11. Слить `n` массивов целых, упорядоченных по возрастанию, в один, упорядоченный по возрастанию. Указание 1. Для упрощения алгоритма следует записать в конец каждого массива барьер – самое большое число соответствующего типа. Барьер, в частности, будет определять, где заканчиваются данные в массиве. Указание 2. Составить функцию, в которую передать динамический двумерный массив (массив одномерных массивов чисел) и число `n` одномерных массивов. Рекомендуется во внешнем цикле завести счетчик одномерных массивов, в которых достигнут барьер. В тот момент, когда он становится равным `n`, слияние окончено.  
Другой вариант: при записи в результирующий массив проверять значение элемента; если оно равно значению барьера, то слияние окончено.



12. Дан массив чисел. Оформить функцию `partition`, перетаскивающую его элементы, удовлетворяющие данному условию, в начало, меняя их местами с предыдущими. Порядок элементов, удовлетворяющих условию, не менять. Условие передавать как параметр-функцию. Например, числовой массив 6 2 9 4 7 3 1 8 5 после применения к нему функции `partition` с условием «*item* > 5» будет выглядеть так: 6 9 7 8 2 4 3 1 5.
13. Дан массив чисел. По нему сконструировать массив сумм соседних элементов, по нему – еще один массив сумм и т.д. – до массива из одного элемента. Результат должен храниться в массиве указателей на одномерные массивы.

### Строки

1. Вводится строка слов, разделенных пробелами (возможны лишние пробелы в начале и в конце строки и между словами). Циклически сдвинуть все слова влево на *k* слов, удалив при этом лишние пробелы (*k* заведомо меньше количества слов).
2. Вводится строка слов, разделенных пробелами (возможны лишние пробелы в начале и в конце строки и между словами). Циклически сдвинуть все слова вправо на *k* слов, удалив при этом лишние пробелы (*k* заведомо меньше количества слов).
3. Вводится строка слов, разделенных пробелами (возможны лишние пробелы в начале и в конце строки и между словами). Скопировать в новую строку два самых коротких слова исходной строки. Алгоритм просмотра исходной строки должен быть однопроходным.
4. Вводится строка слов, разделенных пробелами (возможны лишние пробелы в начале, в конце строки и между словами). Сформировать новую строку, в которой содержатся все слова-перевертыши (палиндромы) исходной строки. Алгоритм просмотра исходной строки должен быть полуторапроходным (полпрохода на проверку того, является ли слово перевертышем).
5. Вводится строка слов, разделенных пробелами (возможны лишние пробелы в начале и в конце строки и между словами), а также целочисленный массив перестановок слов. По данной строке и массиву перестановок сформировать новую строку, удалив при этом лишние пробелы. Например, если задана строка « aa bbb c dd eeee» и массив перестановок 5 2 4 3 1, то итоговая строка должна иметь вид: «eeee bbb dd c aa». Указание: вначале сформировать массив указателей на начала слов.
6. Вводится строка слов, разделенных пробелами (возможны лишние пробелы в начале и в конце строки и между словами). Сформировать строку, в которой слова из исходной строки упорядочены по алфавиту, удалив при этом лишние пробелы. Указание: для сравнения строк можно воспользоваться библиотечной функцией `strcmp(s, s1)`.
7. Вводится строка слов, разделенных пробелами (возможны лишние пробелы в начале, в конце строки и между словами). Сформировать строку, в которой слова из исходной строки упорядочены по длине (а при равной длине порядок их следования остается таким же, как и в исходной строке), удалив при этом лишние пробелы.

8. Вводится строка слов, разделенных пробелами (возможны лишние пробелы в начале, в конце строки и между словами). Сформировать строку, в которой слова из исходной строки упорядочены по количеству гласных (а при равном количестве гласных порядок их следования остается таким же, как и в исходной строке), удалив при этом лишние пробелы.
9. Вводится строка слов, разделенных пробелами (возможны лишние пробелы в начале и в конце строки и между словами). Сформировать строку, в которой удалены лишние пробелы и повторявшиеся ранее слова. Порядок слов не менять.
10. Вводится строка слов, разделенных пробелами (возможны лишние пробелы в начале и в конце строки и между словами). Сформировать строку, в которой слова упорядочены по повторяемости. Дубликаты слов следует удалить. При одинаковой повторяемости первым должно следовать слово, первое вхождение которого встречается раньше в исходной строке.
11. Вводится строка слов, разделенных пробелами (возможны лишние пробелы в начале и в конце строки и между словами). Выдать таблицу слов и количество их повторений в строке. Дубликаты слов не выдавать.
12. Вводится строка. Заменить в ней все цифры их словесными обозначениями: 0 на «zero», 1 на «one», 2 на «two» и т.д.
13. Удалить из строки *s* каждое вхождение подстроки *s1*.
14. Заменить в строке *s* каждое вхождение подстроки *s1* на подстроку *s2*.
15. Даны две строки. Найти индексы (их может быть несколько) и длину самого длинного одинакового участка в обоих массивах. Например, в строках «abracadabra» и «sobrat» самым длинным одинаковым участком будет «bra».
16. Дан массив строк. Сформировать по нему массив подстрок, удовлетворяющих условию, передаваемому в качестве параметра (условие должно представлять собой функцию, принимающую параметр `char *` и возвращающую значение логического типа).
17. Реализовать функцию `char *mystrstr(const char *p, const char *q);` возвращающую первое вхождение подстроки *q* в строку *p* (или 0, если подстрока не найдена).
18. Реализовать функцию `char *mystrpbrk(const char *p, const char *q);` возвращающую указатель на первое вхождение в строку *p* какого-либо символа из строки *q* (или 0, если совпадений не обнаружено).
19. Реализовать функцию `size_t mystrspn(const char *p, const char *q);` возвращающую число начальных символов в строке *p*, которые не совпадают ни с одним из символов из строки *q*.
20. Реализовать функцию `size_t mystrcspn(const char *p, const char *q);` возвращающую число начальных символов в строке *p*, которые совпадают с одним из символов из строки *q*.

21. Реализовать функцию `char* mystrtok(const char *p, const char *q, char* t);` пропускающую символы разделителей, хранящихся в строке `q`, считывающую первую лексему в строке `p` в строку `t` (до следующего символа разделителя или до конца строки) и возвращающую указатель на первый непросмотренный символ.



## **Глава 4**

# **Файловые потоки. Стандартная библиотека C**

Данная глава написана при помощи сайта описывающего не только стандарт C++ но и стандартные библиотеки C/C++ [?]. Огромная благодарность разработчика сайта и людям которые его поддерживают и пишут для него статьи по программированию.

## 4.1 Файловые потоки

Операции ввода/вывода в C++ выполняются также при помощи стандартной библиотеки C – C Standard Input and Output Library `stdio` известной как `stdio.h` в языке C. Эта библиотека используется для работы посредством потоков с физическими устройствами, принтерами, клавиатурой, терминалами и другими типами файлов поддерживаемых операционной системой. Потоки – это некая абстракция для более унифицированной работы с файлами. Все потоки имеют как одинаковые параметры, так и параметры специфичные для устройства с которым они связаны.

Для работы с потоками в стандартной библиотеке `stdio` используется указатель на объект `FILE`. Указатель на `FILE` является уникальным идентификатором потока и используется как параметр для осуществления операций с потоком.

Также, в каждом процессе существуют три стандартных потока: `stdin`, `stderr`, `stdout`, которые автоматически создаются всеми программами которые используют стандартную библиотеку.

Потоки имеют несколько параметров, которые определены для использования некоторыми функциями стандартной библиотекой, а также определяют как следует обрабатывать данные при операциях ввода/вывода. Вот некоторые из таких параметров, доступные после использования функции Открытие файла[`fopen`]:

**Read/Write Access** – Права на чтение запись. Определяет права чтения или записи (или оба) для физического устройства связанного с потоком;

**Text/Binary** – Текстовый или бинарный. Текстовый поток представлен как набор текстовых строк, каждая строка которого заканчивается специальным символом (группой символов) окончания строки. В зависимости от среды, в которой выполняется приложение, некоторые символы (группы символов) могут трансформироваться в специальные символы. В двоичном потоке, при произведении ввода или вывода символов, трансформации не происходит, т.е. символы передаются как есть.

**Buffer** – Буфер. Буфер – это область памяти, где данные собраны перед каждым чтением или записью в связанный с потоком файл или устройство. Потоки могут быть буферизированные или вовсе не иметь буфер. Буферизированные потоки производить чтение или запись в устройство или файл только в случае полного заполнения буфера. Без буфера, в свою очередь, запись или чтение производится на прямую в устройство или файл.

**Orientation** – Направление. При открытии поток не имеет направления. Вскоре, при осуществлении операции ввода/вывода, поток может принять одну из ориентаций: байтовую(`byte-oriented`) или расширенную `wide-oriented`, в зависимости от производимой операции. (как правило, функции, определенные в `<stdio>` являются байт-ориентированным, в то время как функции в `<wchar>` являются широко-ориентированные). См. `swchar` [?] для дополнительной информации. [только C99/C++11]

Потоки также имеют внутренние **индикаторы** которые определяют текущее состояние определенное в зависимости от поведения операций ввода/вывода:

**Error** – Ошибка. Этот индикатор устанавливается при возникновении ошибки, при проведении операции и связана с потоком. Этот индикатор может быть проверен при помощи функции Индикаторы[`ferror`], а также может быть сброшен при вызове функций Индикаторы[`clearerr`], Связывание потока[`freopen`], Позиция в файле[`rewind`].

**End-Of-File** – Окончание файла. Данный индикатор устанавливается после выполнения операции чтения или записи, если достигнут конец файла. Он может быть проверен при помощи функции Индикаторы[`feof`], а также может быть сброшен при вызове функций Индикаторы[`clearerr`] или Связывание потока[`freopen`], а также при выполнении функций смены позиции Позиция в файле[`rewind`], Позиция в файле[`fseek`], Позиция в файле[`fsetpos`].

**Position** – Позиция в файле. Это внутренний указатель, для каждого потока, который указывает на следующий символ для чтения или записи в операциях ввода/вывода. Этот индикатор можно получить при помощи вызовов функции Позиция в файле[`ftell`] и Позиция в файле[`fgetpos`]. Сменить значение индикатора можно при помощи вызова функций Позиция в файле[`rewind`], Позиция в файле[`fseek`], Позиция в файле[`fsetpos`].

## 4.2 Стандартная библиотека C

Рассмотрим основные функции по работе с потоками ввода/вывода.

### 4.2.1 Удаление файла

```
int remove ( const char * filename );
```

Удаляет файл ли директорию имя которого определено в переменной `filename`. Не использует потоки. У пользователя должны быть права на удаление файла.

При удачном удалении файла, функция возвращает 0. При ошибочной работе возвращает любое не нулевое значение и устанавливает переменную `errno`.

```
/* remove example: remove myfile.txt */
#include <stdio>
```

```
int main ()
{
    if( remove( "myfile.txt" ) != 0)
        perror( "Error deleting file" );
    else
        puts( "File successfully deleted" );
    return 0;
}
```

Если файл `myfile.txt` существует перед выполнением программы и пользователь имеет права на удаление, файл будет удален и в стандартный поток вывода будет напечатано:

```
File successfully deleted
```

В другом случае будет напечатано:

```
Error deleting file: No such file or directory
```

### 4.2.2 Переименование файла

```
int rename ( const char * oldname, const char * newname );
```

Переименовывает файл или директорию с именем `oldname` на имя `newname`.

При удачном переименовании файла или директории, функция возвращает 0. При ошибочной работе возвращает любое не нулевое значение и устанавливает переменную `errno`.

```
/* rename example */
#include <stdio>

int main ()
{
    int result;
    char oldname[] = "oldname.txt";
    char newname[] = "newname.txt";
    result= rename( oldname , newname );
    if ( result == 0)
        puts ( "File successfully renamed" );
    else
        perror( "Error renaming file" );
    return 0;
}
```

Если файл `oldname.txt` существует перед выполнением программы и пользователь имеет права на удаление, файл будет переименован и в стандартный поток вывода будет напечатано:

```
File successfully renamed
```

В другом случае будет напечатано:

```
Error renaming file: Permission denied
```

### 4.2.3 Заккрытие файла

```
int fclose ( FILE * stream );
```

Закрывает ранее открытый файл связанный с потоком `stream`. Все внутренние буферы связанные с потоком будут сброшены в связанное с потоком устройство и освобождены.

В случае успеха поток будет закрыт и функция вернет 0 в любом ином случае будет возвращено EOF значение и установлена переменная `errno`.

```
/* fclose example */
#include <stdio>

int main ()
{
    FILE * pFile;
    pFile = fopen ( "myfile.txt", "wt" );
    fprintf ( pFile, "fclose example" );
    fclose ( pFile );
    return 0;
}
```



Этот пример создаст новый файл в текстовом режиме, запишет в него строку «fclose example» и закроет его.

#### 4.2.4 Сброс буфера

```
int fflush ( FILE * stream );
```

Открытый на запись или обновление поток `stream` сбросит внутренний буфер в связанное с ним устройство.

В случае успеха внутренний буфер потока будет записан в связанное с потоком устройство и очищен, а функция вернет 0 в любом ином случае будет возвращено EOF значение. Для получения ошибки можно использовать функцию Индикаторы[`ferror`].

```
/* fflush example */
#include <stdio>

char mybuffer[80];

int main()
{
    FILE * pFile;
    pFile = fopen ("example.txt", "r+");
    if (pFile == NULL) {
        perror ("Error opening file");
    } else {
        fputs ("test", pFile);
        fflush (pFile); // flushing or repositioning required
        fgets (mybuffer, 80, pFile);
        puts (mybuffer);
        fclose (pFile);
        return 0;
    }
}
```

#### 4.2.5 Открытие файла

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

Открывает файл, имя которого определено в переменной `filename` в режиме определенном в переменной `mode`

Режимы:

- "r" Открывает файл только на чтение. Файл должен существовать.
- "w" Создает пустой файл. Если файл существует, его содержимое очищается.
- "a" Открывает файл для добавления. Если файл не существует, создает его. Функции смены позиции Позиция в файле[`fseek`], Позиция в файле[`fsetpos`], Позиция в файле[`rewind`] игнорируются
- "r+" Открывает файл для обновления (чтение и запись). Файл должен существовать.
- "w+" Открывает файл для обновления (чтение и запись). Если файл не существует, создает его.

"a+" Открывает файл для обновления (чтение и запись). Если файл не существует, создает его. Функции смены позиции `fseek`, `fsetpos`, `rewind` можно использовать при следующих операциях.

Дополнительный режим "b" открывает файл в бинарном режиме: "rb", "wb", "ab", "r+b", "w+b", "a+b", "rb+", "wb+", "ab+".

В новом C стандарте C2011 (он не является частью C++) появился новый режим "x", который можно добавлять для любого "w" режима: "wx", "wbx", "w+x" или "w+bx"/"wb+x". Этот модификатор заставляет функцию завершаться с ошибкой, если файл уже существует.

При успешном выполнении функции она возвращает указатель на объект `FILE` (поток), связанный с открываемым устройством.

```
/* fopen example */
#include <stdio>

int main ()
{
    FILE * pFile;
    pFile = fopen ("myfile.txt", "w");
    if (pFile != NULL) {
        fputs ("fopen example", pFile);
        fclose (pFile);
    }
    return 0;
}
```

#### 4.2.6 Связывание потока

`FILE * freopen ( const char * filename, const char * mode, FILE * stream )`  
Связывает существующий поток `stream` с новым файлом определенным в параметре `filename` и с новым режимом определенным в параметре `mode`. Доступные режимы описаны в Открытие файла;

Возвращаемые значения описаны в Открытие файла[`fopen`].

```
/* freopen example: redirecting stdout */
#include <stdio>

int main ()
{
    freopen ("myfile.txt", "w", stdout);
    printf ("This sentence is redirected to a file.");
    fclose (stdout);
    return 0;
}
```

В данном примере стандартный поток вывода `stdout` связывается с файлом `myfile` и записывает в него строку «This sentence is redirected to a file.»

#### 4.2.7 Позиция в файле

```
int fgetpos ( FILE * stream, fpos_t * pos );
```

Функция получения текущего положения в потоке `stream`. Результат помещается в переменную `pos`.

**Возвращаемые значения**

0 – в случае успеха. В ином случае возвращается отличное от нуля значение и устанавливается переменная `errno`.

```
/* fgetpos example */
#include <stdio>

int main ()
{
    FILE * pFile;
    int c;
    int n;
    fpos_t pos;

    pFile = fopen ("myfile.txt", "r");
    if (pFile == NULL) {
        perror ("Error opening file");
    } else {
        c = fgetc (pFile);
        printf ("1st character is %c\n", c);
        fgetpos (pFile, &pos);
        for (n = 0; n < 3; n++) {
            fsetpos (pFile, &pos);
            c = fgetc (pFile);
            printf ("2nd character is %c\n", c);
        }
        fclose (pFile);
    }
    return 0;
}
```

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

Функция установки текущего положения в потоке `stream`. При этом переменная `offset` указывает:

`Binary` – в бинарном режиме количество байт относительно `origin`;

`Text` – 0 или значение возвращенное `Позиция в файле[fte11]`.

В свою очередь переменная `origin` может принимать следующие значения:

`SEEK_SET` – начало файла;

`SEEK_CUR` – текущая позиция;

`SEEK_END` – окончание файла.

### Возвращаемые значения

0 – в случае успеха. В ином случае возвращается отличное от нуля значение. Для получения кода ошибки следует использовать функцию `Индикаторы[ferror]`.

```
/* fseek example */
#include <stdio>

int main ()
{
    FILE * pFile;
    pFile = fopen ( "example.txt" , "wb" );
    fputs ( "This is an apple." , pFile );
    fseek ( pFile , 9, SEEK_SET );
    fputs ( " sam" , pFile );
    fclose ( pFile );
    return 0;
}
```

В результате исполнения в файле `example.txt` будет находиться следующая запись  
`This is a sample.`

```
int fsetpos ( FILE * stream, const fpos_t * pos );
```

Устанавливает позицию pos(ранее полученную при помощи функции Позиция в файле[fgetpos]) в потоке stream.

**Возвращаемые значения**

0 – в случае успеха. В ином случае возвращается отличное от нуля значение и устанавливает переменную errno.

```
/* fsetpos example */
#include <stdio>

int main ()
{
    FILE * pFile;
    fpos_t position;

    pFile = fopen ("myfile.txt", "w");
    fgetpos (pFile, &position);
    fputs ("That is a sample", pFile);
    fsetpos (pFile, &position);
    fputs ("This", pFile);
    fclose (pFile);
    return 0;
}
```

В результате исполнения в файле myfile.txt будет находится следующая запись  
This is a sample.

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

Возвращает текущее положение в потоке stream.

**Возвращаемые значения**

В случае успеха возвращается текущее положение в файле. В ином другом случае будет возвращено значение -1L и установлена переменная errno.

```
/* ftell example : getting size of a file */
#include <stdio>

int main ()
{
    FILE * pFile;
    long size;

    pFile = fopen ("myfile.txt", "rb");
    if (pFile == NULL) {
        perror ("Error opening file");
    } else {
        fseek (pFile, 0, SEEK_END); // non-portable
        size=ftell (pFile);
        fclose (pFile);
        printf ("Size of myfile.txt: %ld bytes.\n", size);
    }
    return 0;
}
```

При удачном исполнении в стандартный поток вывода будет выведен размер файла.

```
void rewind ( FILE * stream );
```

Устанавливает позицию в потоке stream в начало.

```
/* rewind example */
#include <stdio>

int main ()
{
    int n;
    FILE * pFile;
    char buffer [27];

    pFile = fopen ( "myfile.txt", "w+" );
    for ( n = 'A' ; n <= 'Z' ; n++ )
        fputc ( n, pFile );
    rewind (pFile);
    fread (buffer, 1, 26, pFile);
    fclose (pFile);
    buffer[26]='\0';
    puts (buffer);
    return 0;
}
```

В случае удачного исполнения в стандартный поток вывода будет напечатано:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

#### 4.2.8 Чтение/Запись

```
size_t fread ( void * ptr, size_t size, size_t count,
               FILE * stream );
```

Считывает из потока stream, count элементов размера size, в блок памяти на который указывает параметр ptr.

##### Возвращаемые значения

В случае успеха возвращает количество считанных элементов размера size. Если количество считанных элементов отлично от count то следует проверить установленные индикаторы при помощи функций: Индикаторы[ferror], Индикаторы[feof].

```
/* fread example: read an entire file */
#include <stdio>
#include <stdlib>

int main ()
{
    FILE * pFile;
    long lsize;
    char * buffer;
    size_t result;

    pFile = fopen ( "myfile.bin" , "rb" );
    if (pFile==NULL) {
```

```
    fputs ("File error", stderr);
    exit (1);
}

// obtain file size:
fseek (pFile , 0, SEEK_END);
lSize = ftell (pFile);
rewind (pFile);

// allocate memory to contain the whole file:
buffer = (char *) malloc ( sizeof(char) * lSize );
if (buffer == NULL) {
    fputs ("Memory error", stderr);
    exit (2);
}

// copy the file into the buffer:
result = fread (buffer,1,lSize,pFile);
if (result != lSize) {
    fputs ("Reading error", stderr);
    exit (3);
}

/* the whole file is now loaded in the memory buffer. */

// terminate
fclose (pFile);
free (buffer);
return 0;
}
```



```
size_t fwrite ( const void * ptr, size_t size, size_t count,
                FILE * stream );
```

Записывает в поток stream, count элементов размера size, из блока памяти на который указывает параметр ptr.

#### **Возвращаемые значения**

В случае успеха возвращает количество записанных элементов размера size. Если количество считанных элементов отлично от count то следует проверить установленный индикатор при помощи функции: Индикаторы[ferror].

```
/* fwrite example : write buffer */
#include <stdio>

int main ()
{
    FILE * pFile;
    char buffer[] = { 'x' , 'y' , 'z' };
    pFile = fopen ( "myfile.bin" , "wb" );
    fwrite (buffer , 1, sizeof(buffer) , pFile );
    fclose (pFile);
    return 0;
}
```

### **4.2.9 Индикаторы**

```
void clearerr ( FILE * stream );
```

Сбрасывает индикатор ошибки из потока stream.

```
/* writing errors */
#include <stdio>

int main ()
{
    FILE * pFile;
    pFile = fopen("myfile.txt", "r");
    if (pFile == NULL) {
        perror ("Error opening file");
    } else {
        fputc ('x', pFile);
        if ( ferror (pFile) ) {
            printf ("Error writing to myfile.txt\n");
            clearerr (pFile);
        }
        fgetc (pFile);
        if ( !ferror (pFile) )
            printf ("No errors reading myfile.txt\n");
        fclose (pFile);
    }
    return 0;
}
```

```
int feof ( FILE * stream );
```

Проверяет окончание потока stream.

**Возвращаемые значения**

В случае установленного индикатора окончания потока – возвращает не нулевое значение. В ином случае возвращает – 0.

```
/* feof example: byte counter */
#include <stdio>

int main ()
{
    FILE * pFile;
    long n = 0;
    pFile = fopen ("myfile.txt", "rb");
    if (pFile == NULL) {
        perror ("Error opening file");
    } else {
        while ( !feof(pFile) ) {
            fgetc (pFile);
            n++;
        }
        fclose (pFile);
        printf ("Total number of bytes: %d\n", n - 1);
    }
    return 0;
}
```

```
int ferror ( FILE * stream );
```

Функция возвращает код ошибки произошедшей в потоке stream.

**Возвращаемые значения**

Возвращает не нулевое значение – код ошибки. В случае если индикатор ошибки не установлен, возвращает 0.

```
/* ferror example: writing error */
#include <stdio>

int main ()
{
    FILE * pFile;
    pFile = fopen("myfile.txt", "r");
    if (pFile == NULL) {
        perror ("Error opening file");
    } else {
        fputc ('x', pFile);
        if ( ferror (pFile) )
            printf ("Error Writing to myfile.txt\n");
        fclose (pFile);
    }
    return 0;
}
```

#### 4.2.10 Форматированный ввод/вывод

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

Выводит в поток stream данные согласно формата format.

**Формат:**

%[flags][width][.precision][length]specifier Форматированный ввод/вывод

**Возвращаемые значения**

Возвращает количество записанных байт. Если возвращаемое значение отрицательно, код ошибки можно проверить при помощи функции Индикаторы[ferror].

```
/* fprintf example */
#include <stdio>

int main ()
{
    FILE *pFile;
    int n;
    char name[100];

    pFile = fopen ("myfile.txt", "w");
    for (n = 0; n < 3; n++) {
        puts ("please, enter a name: ");
        gets (name);
        fprintf (pFile, "Name %d [%-10.10s]\n", n, name);
    }
    fclose (pFile);
}
```

```
    return 0;  
}
```

<i>specifier</i>	<b>Output</b>	<b>Example</b>
<i>d or i</i>	Signed decimal integer	392
<i>u</i>	Unsigned decimal integer	7235
<i>o</i>	Unsigned octal	610
<i>x</i>	Unsigned hexadecimal integer	7fa
<i>X</i>	Unsigned hexadecimal integer (uppercase)	7FA
<i>f</i>	Decimal floating point, lowercase	392.65
<i>F</i>	Decimal floating point, uppercase	392.65
<i>e</i>	Scientific notation (mantissa/exponent), lowercase	3.9265e+2
<i>E</i>	Scientific notation (mantissa/exponent), uppercase	3.9265E+2
<i>g</i>	Use the shortest representation: <i>%e</i> or <i>%f</i>	392.65
<i>G</i>	Use the shortest representation: <i>%E</i> or <i>%F</i>	392.65
<i>a</i>	Hexadecimal floating point, lowercase	-0xc.90fep-2
<i>A</i>	Hexadecimal floating point, uppercase	-0XC.90FEP-2
<i>c</i>	Character	a
<i>s</i>	String of characters	sample
<i>p</i>	Pointer address	b8000000
<i>n</i>	Nothing printed. The corresponding argument must be a pointer to a signed int. The number of characters written so far is stored in the pointed location.	
<i>%</i>	A <i>%</i> followed by another <i>%</i> character will write a single <i>%</i> to the stream.	%

Рис. 4.1: specifier

<b>flags</b>	<b>description</b>
-	Left-justify within the given field width; Right justification is the default (see <i>width</i> sub-specifier).
+	Forces to precede the result with a plus or minus sign (+ or -) even for positive numbers. By default, only negative numbers are preceded with a - sign.
(space)	If no sign is going to be written, a blank space is inserted before the value.
#	Used with o, x or X specifiers the value is preceeded with 0, 0x or 0X respectively for values different than zero. Used with a, A, e, E, f, F, g or G it forces the written output to contain a decimal point even if no more digits follow. By default, if no digits follow, no decimal point is written.
0	Left-pads the number with zeroes (0) instead of spaces when padding is specified (see <i>width</i> sub-specifier).

Рис. 4.2: flags

<b>width</b>	<b>description</b>
(number)	Minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.
*	The <i>width</i> is not specified in the <i>format</i> string, but as an additional integer value argument preceding the argument that has to be formatted.

Рис. 4.3: width

<b>.precision</b>	<b>description</b>
<b>.number</b>	For integer specifiers (d, i, o, u, x, X): <i>precision</i> specifies the minimum number of digits to be written. If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. A <i>precision</i> of 0 means that no character is written for the value 0. For a, A, e, E, f and F specifiers: this is the number of digits to be printed <b>after</b> the decimal point. For g and G specifiers: This is the maximum number of significant digits to be printed. For s: this is the maximum number of characters to be printed. By default all characters are printed until the ending null character is encountered. If the period is specified without an explicit value for <i>precision</i> , 0 is assumed.
<b>.*</b>	The <i>precision</i> is not specified in the <i>format</i> string, but as an additional integer value argument preceding the argument that has to be formatted.

Рис. 4.4: .precision

	<b>specifiers</b>						
<b>length</b>	d i	u o x X	f F e E g G a A	c	s	p	n
(none)	int	unsigned int	double	int	char*	void*	int*
hh	signed char	unsigned char					signed char*
h	short int	unsigned short int					short int*
l	long int	unsigned long int		wint_t	wchar_t*		long int*
ll	long long int	unsigned long long int					long long int*
j	intmax_t	uintmax_t					intmax_t*
z	size_t	size_t					size_t*
t	ptrdiff_t	ptrdiff_t					ptrdiff_t*
L			long double				

Рис. 4.5: length

### 4.3 Практическая работа

Практическое занятие будет проводится в лабораторном классе. Цели занятия:

1. Получить практические навыки по работе с потоками (файлами);
2. Разобрать вспомогательный код Дополнение к реализации, который в свою очередь будет являться частью последующей курсовой работы Курсовая работа, выполнять роль основного обработчика и обработчика терминального меню;
3. Написать приложение на основе дополнительного кода Дополнение к реализации. Приложение будет реализовывать следующую функциональность: чтение информации из файла, запись информации в файл, формирование информационного заголовка.



## **Глава 5**

# **Потоки ввода/вывода C++**



# **Лабораторные работы**

---

## **Лабораторная работа №1**

## **Лабораторная работа №2**

---

## **Лабораторная работа №3**

## **Лабораторная работа №4**





# **Курсовая работа**

Курсовая работа предназначена для закрепления и углубления знаний студентов по курсу «Программирование 1».

Выполнение данной работы позволяет студентам лучше усвоить основы языка C++ и дает возможность применить теоретические концепции к объектам реального мира.

Разработать программу на языке C++ для обработки данных типа структура с набором следующих операций:

1. создание новой базы данных (с возможностью задания имени файла);
2. просмотр существующей базы данных (чтение данных осуществить из файла);
3. редактирование базы данных (с сохранением изменений в файле);
4. дополнение базы новыми записями (с сохранением в файле);
5. удаление записей из базы данных (с сохранением в файле);
6. поиск в базе данных (по одному и двум поисковым признакам) с выводом на экран найденных записей или сообщения о неуспешном поиске;
7. сортировка данных по заданному полю (с сохранением в файле).

Этапы разработки курсовой работы:

1. Получить задание на выполнение курсовой работы у преподавателя.
2. Разработать техническое задание на разрабатываемую систему.
3. Выполнить анализ задачи.
4. Описать алгоритмы, используемые для решения задачи.
5. Разработать структуру проекта.
6. Разработать и описать интерфейс пользователя.
7. Разработать схему взаимодействия программных единиц, группируя их по модулям.
8. Разработать приложение, реализующее задачу.
9. Разработать пояснительную записку.
10. Создать установочный диск, включающий не только установку программы, но и текст пояснительной записки.
11. Защитить курсовую работу.

Правила оформления пояснительной записки изложены в методических указаниях по курсовому проектированию.

Консультативную помощь по вопросам, связанным с выполнением курсовой работы, можно получить у преподавателя на консультации (согласно графику их проведения).

## Задание

Программа должна иметь текстовый интерфейс. Все операции должны производиться путем выбора, из списка, пункта меню с последующей обработкой данных.

Далее перечислены варианты заданий 5[варианты]. Каждый вариант содержит название предметной области. Студенту необходимо выделить из предметной области абстрактные параметры возможных перечисляемых объектов, для дальнейшего их использования<sup>1</sup>. **Варианты**

1. Паспортный стол.
2. Такси.
3. Школа.
4. Продуктовый магазин.
5. Оружейный склад.
6. Завод.
7. Ферма.
8. Столовая.
9. Военкомат.
10. Кафе.
11. Пожарная охрана.

После выбора задания и определение перечисляемых объектов с абстрактными параметрами, разрабатывается заголовочная (индексная) область файла (базы данных) в которой будет храниться информация о находящихся в файле (базе данных) объектах, а так же **MAGIC number** – специальный набор символов для однозначного определения файла (базы данных). Обычно **MAGIC number** находится в самом начале файла (базы данных) и имеет фиксированный размер.

Исходный код оформляется в соответствии с Правилами оформления кода.

---

<sup>1</sup> Предметная область – медицина, аптека; перечисляемые объекты – лекарства, покупатели, поставщики; лекарство (абстрактные параметры) – год производства, срок годности, производитель, состав, противопоказания

## **Дополнение к реализации**

Листинг 5.1: Заголовочный файл. Файл определений labmore.h

```
1 //
2 // labmore.h
3 // labtemplate
4 //
5 // Created by Khlebnikov Andrey on 09.11.12.
6 // Copyright (c) 2012Khlebinkov Andrey. All rights reserved.
7 //
8
9 #ifndef __labtemplate__labmore__
10 #define __labtemplate__labmore__
11
12 #include <stdio>
13 #include <stdlib>
14 #include <string>
15 #include <locale>
16 #include <errno.h>
17
18 #if !defined(uint8_t) && defined(_MSC_VER)
19 typedef __int8 int8_t;
20 typedef unsigned __int8 uint8_t;
21 typedef __int16 int16_t;
22 typedef unsigned __int16 uint16_t;
23 typedef __int32 int32_t;
24 typedef unsigned __int32 uint32_t;
25 typedef __int64 int64_t;
26 typedef unsigned __int64 uint64_t;
27 #elif !defined(uint8_t)
28 typedef signed char int8_t;
29 typedef unsigned char uint8_t;
30 typedef signed short int16_t;
```

```

31 typedef unsigned short uint16_t;
32 typedef signed int int32_t;
33 typedef unsigned int uint32_t;
34 typedef long long int64_t;
35 typedef unsigned long long uint64_t;
36
37 #endif
38
39 # if defined(_MSC_VER)
40 #pragma warning (disable: 4996)
41 # endif
42
43 #define PRINT_DEBUG_INFO
44
45 #pragma pack(push, 1)
46 struct Index {
47     uint32_t magicNumber;
48     uint32_t counter;
49 };
50
51 struct RecordInfo {
52     size_t itemSize;
53 };
54 #pragma pack(pop)
55 typedef struct Index Index;
56 typedef struct RecordInfo RecordInfo;
57
58 typedef int (*VIEW_FUNCTION_CB)(int nIndex, const void *pMemory, size_t size);
59 typedef int (*UPDATE_FUNCTION_CB)(int nIndex, void *pMemory, size_t size);
60 typedef int (*ADD_FUNCTION_CB)(void *pMemory, size_t size);
61
62 struct tagContext {

```

```

63 FILE *fd;
64 Index index;
65 size_t viewHeight;
66 void *pCurMenu;
67 void *pUserData;
68 char *lpstrFileName;
69 char *lpstrLastError;
70
71 RecordInfo ri;
72 VIEW_FUNCTION_CB view;
73 ADD_FUNCTION_CB add;
74 UPDATE_FUNCTION_CB update;
75 };
76 typedef struct tagContext Context;
77
78 typedef int (*MENU_FUNCTION_CB)(Context *context);
79
80
81 const char *labStdReadString();
82 size_t labStdReadString(const char * const lpstrDescription, char *lpstrBuffer, size_t nBufferSize);
83 const int labStdReadNumber();
84 const int labStdReadNumber(const char * const lpstrDescription);
85 void labFreeString(const void *pMemory);
86 void labStdClearScreen();
87 void labStdPause();
88
89 void *labMenuAppend(void *pMenu, const char *lpstrName, MENU_FUNCTION_CB cb);
90 void *labMenuAppend(void *pMenu, const char *lpstrName, MENU_FUNCTION_CB cb, bool hasShow);
91 void *labMenuAppendSub(void *pMenu, const char *lpstrName, MENU_FUNCTION_CB cb);
92 void *labMenuAppendSub(void *pMenu, const char *lpstrName, MENU_FUNCTION_CB cb, bool hasShow);
93 /**void labMenuAddDefault();*/
94 /**void labMenuPrint();*/

```

```

95 /**void      *labMenuNext(void *pMenu);*/
96 /**void      *labMenuParent(void *pMenu);*/
97 /**int       labMenuEmptyCallback(Context *context);*/
98
99 void          labDatabaseInit(size_t elementSize, uint32_t magicNumber, VIEW_FUNCTION_CB view, ADD_FUNCTION_CB add, U
100 void          labDatabaseSetBegin(Context *context);
101 void          labDatabaseSetEnd(Context *context);
102 bool          labDatabaseIncrementRecord(Context *context);
103 bool          labDatabaseDecrementRecord(Context *context);
104 int           labDatabaseWriteItem(Context *context, const void *lpBuffer, size_t size);
105 bool          labDatabaseReadItem(Context *context, uint32_t nIndex, void *lpBuffer, size_t size);
106 bool          labDatabaseReadNextItem(Context *context, void *lpBuffer, size_t size);
107
108 /**void      labContextSetData(void *pData);*/
109 void          labContextCreateError(Context *context, const char * const lpstrLasError);
110
111 int           labMainLoop();
112
113 #endif /* defined(__labtemplate__labmore__) */

```

Листинг 5.2: Файл реализации labmore.cpp

```

1 //
2 // labmore.cpp
3 // labtemplate
4 //
5 // Created by Khlebnikov Andrey on 09.11.12.
6 // Copyright (c) 2012Khlebnikov Andrey. All rights reserved.
7 //
8
9 #if defined(_WIN32)
10 #include <windows.h>

```



```
11 #else
12 #include <unistd.h>
13 #endif
14 #include <climits>
15 #include "labmore.h"
16
17 struct tagMenuInfo {
18     char *lpstrName;
19     MENU_FUNCTION_CB cb;
20     bool hasShow;
21     struct tagMenuInfo *parent;
22     struct tagMenuInfo *sub;
23     struct tagMenuInfo *next;
24     struct tagMenuInfo *prev;
25 };
26 typedef struct tagMenuInfo MenuInfo;
27
28
29 static MenuInfo *g_menuInfo = NULL;
30 static MenuInfo *g_menuInfoCurrent = NULL;
31 static Context g_context = {
32     NULL, { 0x00000000, 0x00000000 }, 2, NULL, NULL, NULL, NULL, { 0}, NULL, NULL, NULL
33 };
34 static uint32_t g_magicNumber = 0x00000000;
35
36 static int labDestroyContext(Context *context);
37 static int labCloseFile(Context *context);
38 static int labOpenFile(Context *context, bool bCreateIfNotExists);
39 static int labCbCreateDatabase(Context *context);
40 static int labCbExit(Context *context);
41 static int labCbCloseDatabase(Context *context);
```

```

42 static int labCbSettingsSetViewport(Context *context);
43
44 static int labCbAppendRecord(Context *context);
45 static int labCbViewRecord(Context *context);
46 static int labCbUpdateRecord(Context *context);
47 static int labCbRemoveRecord(Context *context);
48
49 static int labMenuSize(MenuInfo *menu);
50 static MenuInfo *labMenuCreate();
51 static MenuInfo *labMenuCreate(const char *lpstrName, MENU_FUNCTION_CB cb);
52 static MenuInfo *labMenuCreate(MenuInfo *parent);
53 static MenuInfo *labMenuCreate(MenuInfo *parent, const char *lpstrName, MENU_FUNCTION_CB cb);
54 static void labMenuDestroy(const MenuInfo *menu);
55 static int labMenuGoPrev(Context *context);
56 static int labMenuGoIndex(Context *context);
57 static MenuInfo *labMenuLast(void *menu);
58 static MenuInfo *labMenuFirst(void *menu);
59 static bool labMenuIsShow(const Context *context, const MenuInfo *menu);
60
61 static void labContextInit(Context *context);
62 static void labContextClearError(Context *context);
63
64 static void labMenuPrint(const Context *context, const MenuInfo *menu);
65
66 static bool labDatabaseWriteRecordsCount(Context *context, uint32_t count);
67
68
69 const char *
70 labStdReadString()
71 {
72     int nAllocatedLength;
73     int nCurrentPos;

```

```
74  int ch;
75  char *lpResult;
76
77  nAllocatedLength = 2;
78  nCurrentPos = 0;
79  lpResult = reinterpret_cast<char *>( malloc(nAllocatedLength) );
80  while (true) {
81      ch = fgetc(stdin);
82      if ( nCurrentPos >= nAllocatedLength - 1) {
83          nAllocatedLength += 1;
84          lpResult = reinterpret_cast<char *>( realloc( lpResult, nAllocatedLength) );
85      }
86      if ( ch == '\n' || ch == '\r' ) {
87          lpResult[nCurrentPos] = 0x00;
88          break;
89      }
90      lpResult[nCurrentPos] = ch;
91      ++nCurrentPos;
92  }
93  return lpResult;
94 }
95
96 void
97 labFreeString(const void *pMemory)
98 {
99     if ( pMemory != NULL)
100         free(const_cast<void *>(pMemory));
101 }
102
103 size_t
104 labStdReadString(const char * const lpstrDescription, char *lpstrBuffer, size_t nBufferSize)
105 {
```

```

106 next:
107     labStdClearScreen();
108     fprintf(stdout, "%s [%lu max length]: ", lpstrDescription == NULL ? "" : lpstrDescription, nBufferSize);
109     const char *lpstrReaded = labStdReadString();
110     size_t nReaded = strlen(lpstrReaded);
111
112     if ( nReaded > nBufferSize ) {
113         fprintf(stdout, "Input more then %lu bytes.\n", nBufferSize);
114         fprintf(stdout, "0\tCat input string\n");
115         fprintf(stdout, "1\tInput new string\n");
116         switch (labStdReadNumber()) {
117             case 1: {
118                 labFreeString(lpstrReaded);
119                 goto next;
120             }
121             default:
122                 break;
123         }
124         nReaded = nBufferSize;
125     }
126     memset(lpstrBuffer, 0, nBufferSize);
127     strncpy(lpstrBuffer, lpstrReaded, nReaded);
128     labFreeString(lpstrReaded);
129     return nReaded;
130 }
131
132 void
133 labStdPause()
134 {
135     #if defined(_WIN32)
136         system("pause");

```

```
137 #else
138 # include <sys/select.h>
139 fd_set readedIn;
140
141 FD_ZERO(&readedIn);
142 FD_SET(fileno(stdin), &readedIn);
143 fprintf(stderr, "Press any key to continue...");
144 select(1, &readedIn, NULL, NULL, NULL);
145 #endif
146 }
147
148 const int
149 labStdReadNumber() {
150     return labStdReadNumber(NULL);
151 }
152
153 const int
154 labStdReadNumber(const char * const lpstrDescription)
155 {
156     const char *pText;
157     int nResult;
158
159     if ( lpstrDescription != NULL )
160         fprintf(stdout, "%s [%d - %d]: ", lpstrDescription, INT_MIN, INT_MAX);
161     nResult = -1;
162     pText = labStdReadString();
163     if ( pText[0] == 0x00 ) {
164         labFreeString(pText);
165         return -1;
166     }
167     nResult = atoi(pText);
168     labFreeString(pText);
169 }
```

```

169     return nResult;
170 }
171
172 #if defined(_WIN32)
173 void
174 labStdClearScreen()
175 {
176     HANDLE hnd1 = GetStdHandle(STD_OUTPUT_HANDLE);
177     CONSOLE_SCREEN_BUFFER_INFO csbi;
178     GetConsoleScreenBufferInfo(hnd1, &csbi);
179     DWORD written;
180     DWORD n = csbi.dwSize.X * csbi.dwCursorPosition.Y + csbi.dwCursorPosition.X + 1;
181     COORD curhome = {0,0};
182     FillConsoleOutputCharacter(hnd1, ' ', n, curhome, &written);
183     csbi.srwindow.Bottom -= csbi.srwindow.Top;
184     csbi.srwindow.Top = 0;
185     SetConsoleWindowInfo(hnd1, TRUE, &csbi.srwindow);
186     SetConsoleCursorPosition(hnd1, curhome);
187 }
188 #elif defined(_POSIX)
189 void
190 labStdClearScreen()
191 {
192     fprintf(stdout, "\e[1;1H\e[2J");
193 }
194 #else
195 #include <term.h>
196 void
197 labStdClearScreen()
198 {
199     system("clear");

```

```
200 }
201 #endif
202
203 int
204 labMenuGoPrev(Context *context)
205 {
206     if ( g_menuInfoCurrent && g_menuInfoCurrent->parent ) {
207         g_menuInfoCurrent = labMenuFirst(g_menuInfoCurrent->parent);
208     }
209     return EXIT_SUCCESS;
210 }
211
212 int
213 labMenuGoIndex(Context *context)
214 {
215     g_menuInfoCurrent = g_menuInfo;
216     return EXIT_SUCCESS;
217 }
218
219 void *
220 labMenuNext(void *pMenu)
221 {
222     if ( pMenu == NULL )
223         return NULL;
224     return static_cast<MenuInfo *>(pMenu)->next;
225 }
226
227 void *
228 labMenuParent(void *pMenu)
229 {
230     if ( pMenu == NULL )
231         return NULL;
```

```
232     return static_cast<MenuInfo *>(pMenu)->parent;
233 }
234
235 void
236 labMenuPrint(const Context *context, MenuInfo *menu)
237 {
238     int iIndex;
239     MenuInfo *next = menu;
240     labStdClearScreen();
241     if ( context->lpstrFileName != NULL && g_context.fid != NULL ) {
242         fprintf(stdout, "[opened : \"%s\"]\n", context->lpstrFileName);
243         fprintf(stdout, "[records: \"%d\"]\n", context->index.counter);
244     }
245     iIndex = 0;
246     while (next != NULL) {
247         if ( labMenuIsShow(context, next) ) {
248             fprintf(stdout, "%d\t%s\n", iIndex++, next->lpstrName);
249         }
250         next = next->next;
251     }
252     fprintf(stdout, "Select menu number: ");
253 }
254
255 void
256 labMenuPrint(const Context *context)
257 {
258     labMenuPrint(context, g_menuInfo);
259 }
260
261 bool
262 labMenuIsShow(const Context *context, const MenuInfo *menu)
```



```
263 {
264     if ( context->fd == NULL ) {
265         return menu->hasShow;
266     }
267     return true;
268 }
269
270 void
271 labMenuDestroy(MenuInfo *menu)
272 {
273     MenuInfo *next;
274     MenuInfo *last;
275
276     next = menu;
277     while ( next != NULL ) {
278         last = next;
279         next = next->next;
280         if ( last->sub != NULL ) {
281             labMenuDestroy(last->sub);
282         }
283         if ( last->lpstrName != NULL ) {
284             #if defined(PRINT_DEBUG_INFO)
285                 fprintf(stdout, "Destroy \"%s\" menu...\n", last->lpstrName);
286             #endif
287             labFreeString(last->lpstrName);
288         }
289         free(last);
290         #if defined(PRINT_DEBUG_INFO)
291             fprintf(stdout, "Destroy success.\n");
292         #endif
293     }
```

```
294 }
295
296 void
297 labMenuAddDefault()
298 {
299     void *pMenu;
300
301     labMenuAppend(NULL, "Exit", labCbExit, true);
302     pMenu = labMenuAppend(NULL, "Settings", NULL, true);
303     labMenuAppendSub(pMenu, "Change viewport", labCbSettingsSetViewport, true);
304     labMenuAppend(NULL, "Create/Open database", labCbCreateDatabase, true);
305     labMenuAppend(NULL, "Add record", labCbAppendRecord);
306     labMenuAppend(NULL, "Update record", labCbUpdateRecord);
307     labMenuAppend(NULL, "View records", labCbViewRecord);
308     labMenuAppend(NULL, "Remove record", labCbRemoveRecord);
309     labMenuAppend(NULL, "Close database", labCbCloseDatabase);
310 }
311
312
313 int
314 labCbCloseDatabase(Context *context) {
315     labCloseFile(context);
316     return EXIT_SUCCESS;
317 }
318
319 MenuInfo *
320 labMenuLast(void *pMenu)
321 {
322     if ( pMenu == NULL )
323         return NULL;
324     MenuInfo *next;
325     MenuInfo *last;
```

```
326
327     next = static_cast<MenuInfo *>(pMenu);
328     while ( next != NULL ) {
329         last = next;
330         next = next->next;
331     }
332     return last;
333 }
334
335 MenuInfo *
336 labMenuFirst(void *pMenu)
337 {
338     if ( pMenu == NULL )
339         return NULL;
340     MenuInfo *prev;
341     MenuInfo *last;
342
343     prev = static_cast<MenuInfo *>(pMenu);
344     while ( prev != NULL ) {
345         last = prev;
346         prev = prev->prev;
347     }
348     return last;
349 }
350
351 void *
352 labMenuAppend(void *pMenu, const char *lpstrName, MENU_FUNCTION_CB cb)
353 {
354     return labMenuAppend(pMenu, lpstrName, cb, false);
355 }
356
357 void *
```

```
358 labMenuAppend(void *pMenu, const char *lpstrName, MENU_FUNCTION_CB cb, bool hasShow)
359 {
360     MenuInfo *result;
361
362     if ( pMenu == NULL ) {
363         if ( g_menuInfo == NULL ) {
364             result = ( g_menuInfo = labMenuCreate(lpstrName, cb) );
365             result->hasShow = hasShow;
366             return result;
367         }
368         pMenu = g_menuInfo;
369     }
370     result = labMenuLast(pMenu);
371     result->next = labMenuCreate(lpstrName, cb);
372     result->next->prev = result;
373     result->next->hasShow = hasShow;
374     return result->next;
375 }
376
377 void *
378 labMenuAppendSub(void *pMenu, const char *lpstrName, MENU_FUNCTION_CB cb)
379 {
380     return labMenuAppendSub(pMenu, lpstrName, cb, false);
381 }
382
383 void *
384 labMenuAppendSub(void *pMenu, const char *lpstrName, MENU_FUNCTION_CB cb, bool hasShow)
385 {
386     MenuInfo *result;
387
388     if ( pMenu == NULL ) {
389         return NULL;
```

```
390 }
391 MenuInfo *m = static_cast<MenuInfo *>(pMenu);
392 if ( m->sub == NULL ) {
393     m->sub = labMenuCreate(m, "Previous", labMenuGoPrev);
394     m->sub->next = labMenuCreate(m, lpstrName, cb);
395     m->sub->next->prev = m->sub;
396     m->sub->hasShow = true;
397     m->sub->next->hasShow = hasShow;
398     return m->sub;
399 }
400 result = labMenuLast( m->sub );
401 result->next = labMenuCreate(m, lpstrName, cb);
402 result->next->prev = result;
403 result->next->hasShow = hasShow;
404 return result->next;
405 }
406
407 int
408 labMenuEmptyCallback(Context *context)
409 {
410     return EXIT_SUCCESS;
411 }
412
413 int
414 labMenuSize(void *menu)
415 {
416     MenuInfo *next = NULL;
417     int count;
418
419     if ( menu == NULL ) {
420         return 0;
421     }
```

```
422     count = 0;
423     next = static_cast<MenuInfo *>(menu)->next;
424     while ( next != NULL ) {
425         next = next->next;
426         ++count;
427     }
428     return count;
429 }
430
431 MenuInfo *
432 labMenuCreate()
433 {
434     MenuInfo *mi = reinterpret_cast<MenuInfo *>( malloc(sizeof(MenuInfo)) );
435     memset(mi, 0, sizeof(MenuInfo));
436     return mi;
437 }
438
439 MenuInfo *
440 labMenuCreate(const char *lpstrName, MENU_FUNCTION_CB cb)
441 {
442     MenuInfo *mi = labMenuCreate();
443     mi->lpstrName = strdup(lpstrName);
444     mi->cb = cb;
445     mi->hasShow = false;
446     return mi;
447 }
448
449 MenuInfo *
450 labMenuCreate(MenuInfo *parent)
451 {
452     MenuInfo *mi = labMenuCreate();
453     mi->parent = parent;
```

```
454     return mi;
455 }
456
457 MenuInfo *
458 labMenuCreate(MenuInfo *parent, const char *lpstrName, MENU_FUNCTION_CB cb)
459 {
460     MenuInfo *mi = labMenuCreate(lpstrName, cb);
461     mi->parent = parent;
462     return mi;
463 }
464
465 int
466 labMainLoop()
467 {
468     int iSelected;
469     int iIndex;
470     int iResult;
471     MenuInfo *next = NULL;
472
473     /** */
474     setlocale(LC_ALL, "Russian");
475
476     labMenuAddDefault();
477     labContextInit(&g_context);
478     g_menuInfoCurrent = g_menuInfo;
479     while (true) {
480         bool bSelected = false;
481         labMenuPrint(&g_context, g_menuInfoCurrent);
482         iSelected = labStdReadNumber();
483         next = g_menuInfoCurrent;
484         iIndex = 0;
```

```
485 while (next != NULL) {
486     if ( labMenuIsShow(&g_context, next) ) {
487         if ( iIndex == iSelected ) {
488             bSelected = true;
489             if ( next->sub != NULL ){
490                 g_menuInfoCurrent = next->sub;
491                 next = NULL;
492                 break;
493             } else {
494                 g_context.pCurMenu = next;
495                 labContextClearError(&g_context);
496                 iResult = (*next->cb)(&g_context);
497                 if ( iResult != EXIT_SUCCESS ) {
498                     if ( g_context.lpstrLastError != NULL ) {
499                         fprintf(stderr, "%s\n", g_context.lpstrLastError);
500                     } else {
501                         fprintf(stderr, "Unknown error.\n");
502                     }
503                     labStdPause();
504                 }
505                 break;
506             }
507         }
508         ++iIndex;
509     }
510     next = next->next;
511 }
512 if ( !bSelected && iSelected != -1 ) {
513     fprintf(stderr, "Unknown menu number: %d\n", iSelected);
514     labStdPause();
515 }
```



```
516     }
517     return EXIT_SUCCESS;
518 }
519
520 int
521 labCbExit(Context *context)
522 {
523     fprintf(stdout, "Program exit.\n");
524     labDestroyContext(context);
525     labMenuDestroy(g_menuInfo);
526     exit(EXIT_SUCCESS);
527     return EXIT_SUCCESS;
528 }
529
530 int
531 labCbSettingsSetViewport(Context *context)
532 {
533     size_t input;
534     #if defined(_MSC_VER)
535     #define FORMAT "ld"
536     #else
537     #define FORMAT "zd"
538     #endif
539     fprintf(stdout, "Input viewport [current %" FORMAT " items]: ", context->viewHeight);
540     input = labStdReadNumber();
541     if (input > 0) {
542         context->viewHeight = input;
543     }
544     return EXIT_SUCCESS;
545 }
546
547 int
```

```
548 labCbViewRecord(Context *context) {
549     void      *pMemory;
550     size_t    nViewCount;
551     bool      complete;
552
553     labContextClearError(context);
554     if ( context->view == NULL ) {
555         labContextCreateError(context, "view function not associated");
556         return EXIT_FAILURE;
557     }
558     complete = false;
559     nViewCount = 1;
560     labStdClearScreen();
561     pMemory = calloc(1, context->ri.itemSize);
562     labDatabaseSetBegin(context);
563     for ( uint32_t nIndex = 0; nIndex < context->index.counter && !complete; ++nIndex ) {
564         if ( labDatabaseReadNextItem(context, pMemory, context->ri.itemSize) ) {
565             (*context->view)( nIndex, pMemory, context->ri.itemSize );
566             fprintf(stdout, "=====\n");
567         } else {
568             free(pMemory);
569             return EXIT_FAILURE;
570         }
571     }
572     if ( ( nViewCount % context->viewHeight ) == 0 && nIndex > 0 ) {
573         if ( nIndex + 1 >= context->index.counter ) {
574             complete = true;
575             labStdPause();
576             continue;
577         }
578         fprintf(stdout, "0. Next items\n");
```

```
579     fprintf(stdout, "1. Complete\n");
580     fprintf(stdout, "Input selected menu: ");
581     int input = labStdReadNumber();
582     switch ( input ) {
583     case 1:
584         complete = true;
585         break;
586     default:
587         labStdClearScreen();
588         break;
589     }
590 }
591 nViewCount++;
592 }
593
594 if ( !complete )
595     labStdPause();
596 free(pMemory);
597 return EXIT_SUCCESS;
598 }
599
600 int
601 labCloseFile(Context *context)
602 {
603     labContextClearError(context);
604     if ( context && context->fd ) {
605         if ( fclose(context->fd) ) {
606             labContextCreateError(context, strerror(errno));
607             return EXIT_FAILURE;
608         }
609         context->fd = NULL;
610         memset(&context->index, 0, sizeof(Index));
```

```
611 }
612
613 if ( context->lpstrFileName != NULL ) {
614     labFreeString(context->lpstrFileName);
615     context->lpstrFileName = NULL;
616 }
617 return EXIT_SUCCESS;
618 }
619
620 int
621 labDestroyContext(Context *context)
622 {
623     labCloseFile(context);
624     return EXIT_SUCCESS;
625 }
626
627 int
628 labCbCreateDatabase(Context *context)
629 {
630     return labOpenFile(context, true);
631 }
632
633 int
634 labOpenFile(Context *context, bool bCreateIfNotExists)
635 {
636     const char *lpstrFileName;
637     int iResult;
638
639     iResult = EXIT_SUCCESS;
640     labContextClearError(context);
641     fprintf(stdout, "Input file name: ");
642     lpstrFileName = labStdReadString();
```

```
643 if ( lpstrFileName[0] != 0x00 ) {
644     if ( context->fd != NULL ) {
645         labCloseFile(context);
646     }
647
648     context->fd = fopen(lpstrFileName, "r+b");
649     if ( context->fd == NULL && bCreateIfNotExists ) {
650         context->fd = fopen(lpstrFileName, "w+b");
651     }
652
653     if ( context->fd == NULL ) {
654         labContextCreateError(context, strerror(errno));
655         iResult = EXIT_FAILURE;
656     } else {
657         if ( context->lpstrFileName != NULL ) {
658             labFreeString(context->lpstrFileName);
659             lpstrFileName = NULL;
660         }
661         /** NOTICE: File open success */
662         fpos_t pos;
663
664         pos = 0;
665         // Move to end of file
666         fseek(context->fd, 0, SEEK_END);
667         // Get file position
668         fgetpos(context->fd, &pos);
669         // Move to file begin
670         rewind(context->fd);
671         if ( pos == 0 ) {
672             /** NOTICE: Init database header */
673             context->index.counter = 0L;
```

```
674     fwrite( &context->index, sizeof(Index), 1, context->fd );
675     context->lpstrFileName = strdup(lpstrFileName);
676     iResult = EXIT_SUCCESS;
677 } else if ( pos >= sizeof(Index) ) {
678     rewind( context->fd );
679     fread(&context->index, sizeof(Index), 1, context->fd);
680     if (context->index.magicNumber == g_magicNumber) {
681         context->lpstrFileName = strdup(lpstrFileName);
682     } else {
683         labCloseFile(context);
684         labContextCreateError(context, "Unknwon file Magic Number.");
685         iResult = EXIT_FAILURE;
686     }
687 } else {
688     labContextCreateError(context, "Unknwon file type.");
689     iResult = EXIT_FAILURE;
690 }
691 }
692 } else {
693     labContextCreateError(context, "Illegal file name.");
694     iResult = EXIT_FAILURE;
695 }
696 labFreeString(lpstrFileName);
697 return iResult;
698 }
699
700
701 void
702 labContextSetData(void *pUserData)
703 {
704     g_context.pUserData = pUserData;
```

```
705 }
706
707 void
708 labContextInit(Context *context)
709 {
710     context->fd = NULL;
711     context->index.counter = 0L;
712     context->pCurMenu = NULL;
713     context->lpstrFileName = NULL;
714     context->lpstrLastError = NULL;
715 }
716
717 void
718 labContextClearError(Context *context)
719 {
720     if ( context->lpstrLastError != NULL ) {
721         labFreeString(context->lpstrLastError);
722     }
723     context->lpstrLastError = NULL;
724 }
725
726 void
727 labContextCreateError(Context *context, const char * const lpstrLastError)
728 {
729     labContextClearError(context);
730     context->lpstrLastError = strdup(lpstrLastError);
731 }
732
733 void
734 labDatabaseSetBegin(Context *context)
735 {
736     if ( context->fd == NULL )
```

```
737     return;
738     fseek( context->fd, sizeof(Index), SEEK_SET );
739 }
740
741 void
742 labDatabaseSetEnd(Context *context)
743 {
744     fpos_t pos;
745     if ( context->fd == NULL )
746         return;
747     labDatabaseSetBegin(context);
748     fgetpos( context->fd, &pos );
749     pos += context->index.counter * context->ri.itemSize;
750     fsetpos( context->fd, &pos );
751 }
752
753 bool
754 labDatabaseWriteRecordsCount(Context *context, uint32_t count) {
755     fpos_t pos;
756     size_t nwrote;
757     uint32_t oldCount;
758     bool bResult;
759
760     bResult = false;
761     if ( context->fd == NULL )
762         return bResult;
763
764     oldCount = context->index.counter;
765     context->index.counter = count;
766     pos = 0L;
767     fsetpos( context->fd, &pos );
768     nwrote = fwrite( &context->index, sizeof(Index), 1, context->fd );
```



```
769     if ( nwrited != 1) {
770         context->index.counter = oldCount;
771     } else {
772         bResult = true;
773     }
774     fsetpos( context->fd, &pos );
775     return bResult;
776 }
777
778 bool
779 labDatabaseIncrementRecord(Context *context)
780 {
781     return labDatabaseWriteRecordsCount( context, context->index.counter + 1);
782 }
783
784 bool
785 labDatabaseDecrementRecord(Context *context)
786 {
787     return labDatabaseWriteRecordsCount( context, context->index.counter - 1);
788 }
789
790 int
791 labDatabaseWriteItem(Context *context, const void *lpBuffer, size_t size)
792 {
793     size_t      nwrited;
794
795     if ( context->fd == NULL )
796         return false;
797     labDatabaseSetEnd(context);
798     nwrited = fwrite( static_cast<uint8_t *>(const_cast<void *>(lpBuffer)), size, 1, context->fd );
799     if ( nwrited == 1) {
800         if ( !labDatabaseIncrementRecord(context) ) {
```

```

801     labContextCreateError(context, "Can't increment database records.");
802     return EXIT_FAILURE;
803 }
804 } else {
805     labContextCreateError(context, "Can't write record to database.");
806     return EXIT_FAILURE;
807 }
808 return EXIT_SUCCESS;
809 }
810
811 bool
812 labDatabaseReadNextItem(Context *context, void *lpBuffer, size_t size)
813 {
814     size_t nReaded;
815     if ( context->fd == NULL )
816         return false;
817     if ( feof(context->fd) ) {
818         labContextCreateError(context, "End of file");
819         return false;
820     }
821     nReaded = fread( lpBuffer, size, 1, context->fd );
822     if ( nReaded != 1 ) {
823         labContextCreateError(context, "Error read buffer");
824         return false;
825     }
826     return true;
827 }
828
829 bool
830 labDatabaseReadItem(Context *context, uint32_t nIndex, void *lpBuffer, size_t size)
831 {

```

```
832 fpos_t pos;
833 if ( context->fd == NULL )
834     return false;
835 if ( context->index.counter <= nIndex ) {
836     labContextCreateError(context, "Illegal item index");
837     return false;
838 }
839 labDatabaseSetBegin(context);
840 fgetpos(context->fd, &pos);
841 pos += nIndex * size;
842 fsetpos(context->fd, &pos);
843 return labDatabaseReadNextItem(context, lpBuffer, size);
844 }
845
846 int
847 labCbAppendRecord(Context *context) {
848     void *pMemory;
849     int iResult;
850
851     labContextClearError(context);
852     if ( context->add == NULL ) {
853         labContextCreateError(context, "Add callback function not associated");
854         return EXIT_FAILURE;
855     }
856     pMemory = calloc(1, context->ri.itemSize);
857     iResult = (*context->add)(pMemory, context->ri.itemSize);
858     if ( iResult != EXIT_FAILURE ) {
859         iResult = labDatabaseWriteItem( context, pMemory, context->ri.itemSize );
860     }
861     free(pMemory);
862     return iResult;
```

```
863 }
864
865 int
866 labCbRemoveRecord(Context *context)
867 {
868     fpos_t pos;
869     int nIndex;
870     int iResult;
871
872     iResult = EXIT_SUCCESS;
873     fprintf(stdout, "Input remove element[0 - %d]: ", context->index.counter - 1);
874     nIndex = labStdReadNumber();
875     if ( nIndex < 0 || (uint32_t)nIndex >= context->index.counter ) {
876         iResult = EXIT_FAILURE;
877         labContextCreateError(context, "Illegal element index");
878     } else {
879
880         if ( context->index.counter != 1 ) {
881             void *pMemory = calloc(1, context->ri.itemSize );
882             labDatabaseSetBegin(context);
883             fgetpos( context->fd, &pos );
884             pos += nIndex * context->ri.itemSize;
885             for ( int i = nIndex + 1; (uint32_t)i < context->index.counter; ++i ) {
886                 pos += context->ri.itemSize;
887                 fsetpos( context->fd, &pos );
888                 fread( pMemory, context->ri.itemSize, 1, context->fd );
889                 fgetpos( context->fd, &pos );
890                 pos -= context->ri.itemSize;
891                 pos -= context->ri.itemSize;
892                 fsetpos( context->fd, &pos );
893                 fwrite( pMemory, context->ri.itemSize, 1, context->fd );
```

```
894     fgetpos( context->fd, &pos );
895 }
896 free(pMemory);
897 }
898 labDatabaseDecrementRecord(context);
899 }
900 return iResult;
901 }
902
903 int
904 labCbUpdateRecord(Context *context)
905 {
906     void *pMemory;
907     int iResult;
908     int nIndex;
909
910     labContextClearError(context);
911     if ( context->update == NULL ) {
912         labContextCreateError(context, "Add callback function not associated");
913         return EXIT_FAILURE;
914     }
915     iResult = EXIT_SUCCESS;
916     pMemory = calloc(1, context->ri.itemSize);
917     fprintf(stdout, "Input updated element[0 - %d]: ", context->index.counter - 1);
918     nIndex = labStdReadNumber();
919     if ( nIndex < 0 || (uint32_t)nIndex >= context->index.counter ) {
920         iResult = EXIT_FAILURE;
921         labContextCreateError(context, "Illegal element index");
922     } else {
923         void *pMemory = calloc(1, context->ri.itemSize);
924         if ( labDatabaseReadItem(context, nIndex, pMemory, context->ri.itemSize) ) {
```

```

925     fpos_t pos;
926     (*context->update)(nIndex, pMemory, context->ri.itemSize);
927     labDatabaseSetBegin(context);
928     fgetpos( context->fd, &pos );
929     pos += context->ri.itemSize * nIndex;
930     fsetpos( context->fd, &pos );
931     fwrite(pMemory, context->ri.itemSize, 1, context->fd );
932 } else {
933     iResult = EXIT_FAILURE;
934 }
935 free(pMemory);
936 }
937 free(pMemory);
938 return iResult;
939 }
940
941 void
942 labDatabaseInit(size_t elementSize, uint32_t magicNumber, VIEW_FUNCTION_CB view, ADD_FUNCTION_CB add, UPDATE_FU
943 {
944     g_context.ri.itemSize = elementSize;
945     g_context.view = view;
946     g_context.add = add;
947     g_context.update = update;
948     g_magicNumber = magicNumber;
949     g_context.index.magicNumber = magicNumber;
950 }

```

# Приложение

## Вопросы к зачету

1. Что такое язык C++?
2. Парадигмы программирования.
3. Процедурное программирование.
4. Модульное программирование.
5. Абстракция данных.
6. Обобщенное программирование.
7. Пространство имен стандартной библиотеки.
8. Вывод.
9. Строки.
10. Ввод.
11. Алгоритмы.
12. Типы.
13. Логический тип.
14. Символьные типы.
15. Целые типы.
16. Типы с плавающей запятой.
17. Размеры.
18. Перечисления.
19. Объявления.
20. Указатели.
21. Массивы.
22. Указатели на массивы.
23. Константы.
24. Ссылки.
25. Структуры.
26. Калькулятор.
27. Обзор операций языка C++.
28. Обзор операторов языка C++.
29. Объявление функций.
30. Передача аргументов.



31. Возвращаемое значение.
32. Перегрузка имен функций.
33. Аргументы по умолчанию.
34. Макросы.
35. Разбиение на модули и интерфейсы.
36. Пространства имен.
37. Исключения.
38. Раздельная компиляция.
39. компоновка.
40. Программы.
41. Проектные решения стандартной библиотеки.

**Правила оформления практических заданий**

**Пример оформленного задания**

**Правила оформления лабораторной работы**

**Правила оформления курсовой работы**

**Пример оформленной курсовой работы**

**Правила оформления кода**

### Отступы

- \* Для обозначения отступа используйте 4 пробела подряд;
- \* Используйте пробелы, а не табуляцию.

### Объявление переменных

- \* Объявляйте по одной переменной в строке;
- \* Избегайте, если это возможно, коротких и запутанных названий переменных (Например: «a», «rbarr», «nughdeget»);
- \* Односимвольные имена переменных подходят только для итераторов циклов, небольшого локального контекста и временных переменных. В остальных случаях имя переменной должно отражать ее назначение;
- \* Заводите переменные только по мере необходимости:

```
// Wrong
int a, b;
char *c, *d;
// Correct
int height;
int width;
char *nameOfThis;
char *nameOfThat;
```

- \* Функции и переменные должны именоваться с прописной буквы, а если имя переменной или функции состоит из нескольких слов, то первое слово должно начинаться с прописной буквы, остальные – со строчных;
- \* Избегайте аббревиатур:

```
// Wrong
short Cntr;
char ITEM_DELIM = '\t';
// Correct
short counter;
char itemDelimiter = '\t';
```

- \* Имена классов всегда начинаются с заглавной буквы.

### Пробелы

- \* Используйте пустые строки для логической группировки операторов, где это возможно;
- \* Всегда используйте одну пустую строку в качестве разделителя;
- \* Всегда используйте один пробел перед фигурной скобкой:

```
// Wrong
if(foo){
}

// Correct
if (foo) {
}
```

- \* Всегда ставьте один пробел после \* или &, если они стоят перед описанием типов. Но никогда не ставьте пробелы после \* или & и именем переменной:

```
char *x;
const Class &myClass;
const char * const y = "hello";
```

- \* Бинарные операции отделяются пробелами с 2-х строн;
- \* После преобразования типов не ставьте пробелов;
- \* Избегайте преобразования типов в стиле C:

- \*

```
// Wrong
char * blockOfMemory = (char *) malloc(data.size());
// Correct
char * blockOfMemory = reinterpret_cast<char *>( malloc(data.size()));
```

### Фигурные скобки

- \* Возьмите за основу расстановку открывающих фигурных скобок на одной строке с выражением, которому они предшествуют:

```
// Wrong
if (codec)
{
}

// Correct
if (codec) {
}
```

- \* Исключение: Тело функции и описание класса всегда открывается фигурной скобкой, стоящей на новой строке:

```
static void foo(int g)
{
    fprintf(stdout, "foo: %i", g);
}

class Moo
{
};
```

- \* Используйте фигурные скобки в условиях, если тело условия в размере превышает одну линию, или тело условия достаточно сложное и выделение скобками действительно необходимо:

```
// Wrong
if (address.isEmpty()) {
    return false;
}

for (int i = 0; i < 10; ++i) {
    fprintf(stdout, "%i", i);
}

// Correct
if (address.isEmpty())
    return true;

for (int i = 0; i < 10; ++i)
    fprintf(stdout, "%i", i);
```

- \* Исключение 1: Используйте скобки, если родительское выражение состоит из нескольких строк или оберток:

```
// Correct
if (address.isEmpty() || !isValid()
    || !codec) {
    return false;
}
```

- \* Исключение 2: Используйте фигурные скобки, когда тела ветвлений if-then-else занимают несколько строчек:

```
// Wrong
if (address.isEmpty())
    return false;
else {
    fprintf(stdout, "%s", address.c_str());
    ++it;
}

// Correct
if (address.isEmpty()) {
    return false;
} else {
    fprintf(stdout, "%s", address.c_str());
    ++it;
}

// Wrong
if (a)
    if (b)
        ...
    else
```

```
...  
  
// Correct  
// Wrong  
if (a) {  
    if (b)  
        ...  
    else  
        ...  
}
```

- \* Используйте фигурные скобки для обозначения пустого тела условия:

```
// Wrong  
while (a);  
  
// Correct  
while (a) {}
```

### Круглые скобки

- \* Используйте круглые скобки для группировки выражений:

```
// Wrong  
if (a && b || c)  
  
// Correct  
if ((a && b) || c)  
  
// Wrong  
a + b & c  
// Correct  
(a + b) & c
```

### Использование конструкции switch

- \* Операторы case должны быть в одном столбце со switch
- \* Каждый оператор case должен иметь закрывающий break (или return) или комментарий, котрой предполагает намеренное отсутствие break или return:

```
switch (myEnum) {  
case value1:  
    doSomething();  
    break;  
case value2:  
    doSomethingElse();  
    // continue  
default:  
    defaultHandling();  
    break;  
}
```

### Разрыв строк

- \* Длина строки кода не должна превышать 100 символов. Если надо – используйте разрыв строки.
- \* Запятые помещаются в конец разорванной линии; операторы помещаются в начало новой строки. В зависимости от используемой вами IDE, оператор на конце разорванной строки можно проглядеть:

```
// Correct
if (longExpression
    + otherLongExpression
    + otherOtherLongExpression) {
}

// Wrong
if (longExpression +
    otherLongExpression +
    otherOtherLongExpression) {
}
```

### Наследование и ключевое слово `virtual`

- \* При переопределении `virtual`-метода, ни за что не помещайте слово `virtual` в заголовочный файл.

### Главное исключение

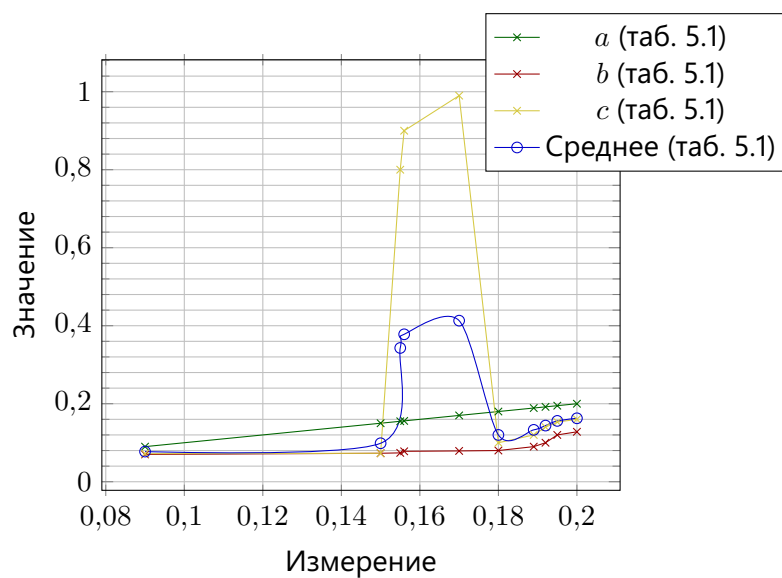
- \* Не бойтесь нарушать описанные выше правила, если вам кажется, что они только запутают ваш код.

## **Графики**



Таблица 5.1: Средние числа

Значение $a$	Значение $b$	Значение $c$	Среднее значение
0,09	0,07	0,072	0,077333
0,15	0,073	0,073	0,098667
0,155	0,074	0,8	0,343
0,156	0,078	0,9	0,378
0,17	0,079	0,99	0,413
0,18	0,08	0,1	0,12
0,189	0,09	0,12	0,133
0,192	0,1	0,14	0,144
0,195	0,12	0,153	0,156
0,2	0,128	0,16	0,162667



**Таблица ASCII**

DEC	OCT	HEX	BIN	Symbol	HTML Number	HTML Name	Description
0	000	00	00000000	NUL	&#000;		Null char
1	001	01	00000001	SOH	&#001;		Start of Heading
2	002	02	00000010	STX	&#002;		Start of Text
3	003	03	00000011	ETX	&#003;		End of Text
4	004	04	00000100	EOT	&#004;		End of Transmission
5	005	05	00000101	ENQ	&#005;		Enquiry
6	006	06	00000110	ACK	&#006;		Acknowledgment
7	007	07	00000111	BEL	&#007;		Bell
8	010	08	00001000	BS	&#008;		Back Space
9	011	09	00001001	HT	&#009;		Horizontal Tab
10	012	0A	00001010	LF	&#010;		Line Feed
11	013	0B	00001011	VT	&#011;		Vertical Tab
12	014	0C	00001100	FF	&#012;		Form Feed
13	015	0D	00001101	CR	&#013;		Carriage Return
14	016	0E	00001110	SO	&#014;		Shift Out / X-On
15	017	0F	00001111	SI	&#015;		Shift In / X-Off
16	020	10	00010000	DLE	&#016;		Data Line Escape
17	021	11	00010001	DC1	&#017;		Device Control 1 (oft. XON)
18	022	12	00010010	DC2	&#018;		Device Control 2
19	023	13	00010011	DC3	&#019;		Device Control 3 (oft. XOFF)
20	024	14	00010100	DC4	&#020;		Device Control 4
21	025	15	00010101	NAK	&#021;		Negative Acknowledgement
22	026	16	00010110	SYN	&#022;		Synchronous Idle
23	027	17	00010111	ETB	&#023;		End of Transmit Block
24	030	18	00011000	CAN	&#024;		Cancel
25	031	19	00011001	EM	&#025;		End of Medium
26	032	1A	00011010	SUB	&#026;		Substitute
27	033	1B	00011011	ESC	&#027;		Escape

28	034	1C	00011100	FS	&#028;		File Separator
29	035	1D	00011101	GS	&#029;		Group Separator
30	036	1E	00011110	RS	&#030;		Record Separator
31	037	1F	00011111	US	&#031;		Unit Separator
32	040	20	00100000		&#32;		Space
33	041	21	00100001	!	&#33;		Exclamation mark
34	042	22	00100010	"	&#34;	&quot;	Double quotes (or speech marks)
35	043	23	00100011	#	&#35;		Number
36	044	24	00100100	\$	&#36;		Dollar
37	045	25	00100101	%	&#37;		Procenttecken
38	046	26	00100110	&	&#38;	&amp;	Ampersand
39	047	27	00100111	'	&#39;	&apos;	Single quote
40	050	28	00101000	(	&#40;		Open parenthesis (or open bracket)
41	051	29	00101001	)	&#41;		Close parenthesis (or close bracket)
42	052	2A	00101010	*	&#42;		Asterisk
43	053	2B	00101011	+	&#43;		Plus
44	054	2C	00101100	,	&#44;		Comma
45	055	2D	00101101	-	&#45;		Hyphen
46	056	2E	00101110	.	&#46;		Period, dot or full stop
47	057	2F	00101111	/	&#47;		Slash or divide
48	060	30	00110000	0	&#48;		Zero
49	061	31	00110001	1	&#49;		One
50	062	32	00110010	2	&#50;		Two
51	063	33	00110011	3	&#51;		Three
52	064	34	00110100	4	&#52;		Four
53	065	35	00110101	5	&#53;		Five
54	066	36	00110110	6	&#54;		Six
55	067	37	00110111	7	&#55;		Seven
56	070	38	00111000	8	&#56;		Eight

57	071	39	00111001	9	&#57;		Nine
58	072	3A	00111010	:	&#58;		Colon
59	073	3B	00111011	;	&#59;		Semicolon
60	074	3C	00111100	<	&#60;	&lt;	Less than (or open angled bracket)
61	075	3D	00111101	=	&#61;		Equals
62	076	3E	00111110	>	&#62;	&gt;	Greater than (or close angled bracket)
63	077	3F	00111111	?	&#63;		Question mark
64	100	40	01000000	@	&#64;		At symbol
65	101	41	01000001	A	&#65;		Uppercase A
66	102	42	01000010	B	&#66;		Uppercase B
67	103	43	01000011	C	&#67;		Uppercase C
68	104	44	01000100	D	&#68;		Uppercase D
69	105	45	01000101	E	&#69;		Uppercase E
70	106	46	01000110	F	&#70;		Uppercase F
71	107	47	01000111	G	&#71;		Uppercase G
72	110	48	01001000	H	&#72;		Uppercase H
73	111	49	01001001	I	&#73;		Uppercase I
74	112	4A	01001010	J	&#74;		Uppercase J
75	113	4B	01001011	K	&#75;		Uppercase K
76	114	4C	01001100	L	&#76;		Uppercase L
77	115	4D	01001101	M	&#77;		Uppercase M
78	116	4E	01001110	N	&#78;		Uppercase N
79	117	4F	01001111	O	&#79;		Uppercase O
80	120	50	01010000	P	&#80;		Uppercase P
81	121	51	01010001	Q	&#81;		Uppercase Q
82	122	52	01010010	R	&#82;		Uppercase R
83	123	53	01010011	S	&#83;		Uppercase S
84	124	54	01010100	T	&#84;		Uppercase T
85	125	55	01010101	U	&#85;		Uppercase U

86	126	56	01010110	V	&#86;		Uppercase V
87	127	57	01010111	W	&#87;		Uppercase W
88	130	58	01011000	X	&#88;		Uppercase X
89	131	59	01011001	Y	&#89;		Uppercase Y
90	132	5A	01011010	Z	&#90;		Uppercase Z
91	133	5B	01011011	[	&#91;		Opening bracket
92	134	5C	01011100		&#92;		Backslash
93	135	5D	01011101	]	&#93;		Closing bracket
94	136	5E	01011110	^	&#94;		Caret - circumflex
95	137	5F	01011111	_	&#95;		Underscore
96	140	60	01100000	'	&#96;		Grave accent
97	141	61	01100001	a	&#97;		Lowercase a
98	142	62	01100010	b	&#98;		Lowercase b
99	143	63	01100011	c	&#99;		Lowercase c
100	144	64	01100100	d	&#100;		Lowercase d
101	145	65	01100101	e	&#101;		Lowercase e
102	146	66	01100110	f	&#102;		Lowercase f
103	147	67	01100111	g	&#103;		Lowercase g
104	150	68	01101000	h	&#104;		Lowercase h
105	151	69	01101001	i	&#105;		Lowercase i
106	152	6A	01101010	j	&#106;		Lowercase j
107	153	6B	01101011	k	&#107;		Lowercase k
108	154	6C	01101100	l	&#108;		Lowercase l
109	155	6D	01101101	m	&#109;		Lowercase m
110	156	6E	01101110	n	&#110;		Lowercase n
111	157	6F	01101111	o	&#111;		Lowercase o
112	160	70	01110000	p	&#112;		Lowercase p
113	161	71	01110001	q	&#113;		Lowercase q
114	162	72	01110010	r	&#114;		Lowercase r

115	163	73	01110011	s	&#115;		Lowercase s
116	164	74	01110100	t	&#116;		Lowercase t
117	165	75	01110101	u	&#117;		Lowercase u
118	166	76	01110110	v	&#118;		Lowercase v
119	167	77	01110111	w	&#119;		Lowercase w
120	170	78	01111000	x	&#120;		Lowercase x
121	171	79	01111001	y	&#121;		Lowercase y
122	172	7A	01111010	z	&#122;		Lowercase z
123	173	7B	01111011	{	&#123;		Opening brace
124	174	7C	01111100		&#124;		Vertical bar
125	175	7D	01111101	}	&#125;		Closing brace
126	176	7E	01111110		&#126;		Equivalency sign - tilde
127	177	7F	01111111		&#127;		Delete