Семен Игонин

Старт с C++

Саратов, 2023

Книга для тех, кто стремится начать путь программиста с изучения самого сложного, но полезного языка программирования C++. Рассматривается не столько конкретный язык, сколько общие принципы разработки программ. Особое внимание уделяется низкоуровневым нюансам. Понимание низкоуровневых процессов, того как выполняется программа компьютером, позволит писать эффективный высокоуровневый код. Книга написана учителем и не претендует на академическую точность. Однако, для первого знакомства с программированием может оказаться полезной.

Книга, представленная электронной версией, распространяется свободно и бесплатно.

Рецензенты и редакторы: *вакантная позиция, буду премного благодарен за любую помощь в указании на грамматические, стилистические и логические ошибки. Исправления выделять жёлтым цветом и присылать на is-info64@ya.ru, отзывы и пожелания туда же.*

Ссылка на .odt версию для редактирования: https://disk.yandex.ru/d/ehZb6LB6eewe1w

Содержание

Предисловие

Программирование — навык, который следует в себе развивать не смотря на все сложности, связанные с этим процессом. Самой популярной отговоркой является - «зачем мне программирование, я же не программист».

Данный навык важен, поскольку в процессе обучения программированию развиваются следующие сверхъестественные способности:

- логика
- алгоритмическое мышление
- способность к структурированию и систематизированию информации
- умение устанавливать причинно следственные связи
- умение формализовывать условия задачи
- умение разбивать сложные нерешаемые задачи на простые решаемые

Жизнь не всегда логична и планируема, не для всего возможно составить алгоритм. Но многие из аспектов современного мира связаны с интенсивной активацией интеллектуальных способностей в сферах, часто, далёких от программирования в техническом смысле. Однако, «сверхъестественные способности» являются важной частью становления индивидуума, как интеллектуально развитой личности. Программирование является запросом цивилизации для обеспечения своего дальнейшего развития.

Программирование в современном мире, можно сравнить с такими навыками, как умение читать или считать. Прочитав книгу, читать или считать не научишься. К счастью, с программированием также. Приобретение любого навыка реализуется только через практику, усердие, неудачи, слезы и победы.

Для кого?

Изложение материала ориентировано на людей, не сталкивающихся с программированием, но имеющих стремление развиваться в данной области. Первые семь глав могут быть полезны в качестве учебного пособия для школьников. Оставшиеся главы больше для тех, кто планирует заняться программированием по серьёзному, либо хочет ознакомиться с фундаментом IT технологий для общего развития.

Проще, это книга для тех, кто хочет стать программистом.

О чем?

Цель книги: заложить фундамент, на котором возможно выстроить правильную стратегию изучения любого языка программирования. Базовые вещи разбираются с особой тщательностью. Уделяется много времени не только описанию алгоритмов, но и рассматривается связь программы с железом. Поясняется, что происходит при объявлении переменных и при чем тут типы.

В высокоуровневом плане рассматриваются основные алгоритмические конструкции:

- следование,
- ветвление,
- циклы,

и структуры данных:

- массивы.

Алгоритмы + Структуры данных = Программы

На первом этапе обучения, важно понимание общих принципов исполнения программы компьютером, развитие навыков формализации и составления алгоритмов. Изучение какого-то конкретного языка программирования вторично.

Можно было бы отказаться от языка вовсе и составлять алгоритмы в виде блок-схем или псевдокода, данные приёмы будут широко применяться.

С практической же точки зрения, знание одного из языков может сильно пригодиться при решении рабочих задач даже для профессий, далёких от программирования. Любые процессы, связанные с обработкой или хранением данных, могут потребовать знаний языков программирования. Автоматизация с использованием разработанных вами программ может сэкономить часы, недели, а то и месяцы рабочего времени. Ярким примером является язык программирования VisualBasic for Application (VBA), позволяющий автоматизировать обработку данных в таких редакторах как MS Excel, MS Word.

Чем более язык высокоуровневый, тем он проще. Однако с использованием таких сверхвысокоуровневых языков изучать программирование сложнее. Сверхвысокоуровневые языки очень многое скрывают. Чтобы понять те или иные сверхвысокоуровневые конструкции, всё равно приходится спускаться вниз, к языкам программирования C и C++.

Так не лучше ли с них и начать, чтобы не обманывать себя.

Выбор языка программирования

Стандартом программирования давно является язык C и его потомок C++. Для достижения поставленной цели он подходит как нельзя лучше. В языке имеется синтаксис, направленный на развитие алгоритмического мышления и способности к структурированию информации. Наличие типов данных и указателей позволяет понять, как электронно-вычислительная машина (компьютер) выполняет программу (работает). Более того, множество языков программирования, среди которых Java, C#, PHP имеют схожий синтаксис.

В книге рассматривается основные понятия и конструкции языка C++:

- настройка среды разработки, написание и запуск первой программы
- переменные и типы данных
- ввод/вывод на консоль,
- алгоритмические конструкции: следование, ветвление, циклы,
- массивы,
- функции.

Перечисленные разделы являются минимальным набором, для тех, кто стремится научится писать простые программы.

Начиная с главы 8 происходит более углубленное погружение:

- указатели
- ссылки
- структуры
- шаблоны
- перегрузка операторов
- компиляция — подробный разбор
- модульный подход и пространства имён

Данные разделы предназначены для тех, кто стремится изучать программирование серьёзно. Либо кому интересно, что происходит на более низком уровне. Или кому понравилась книга, и он читает её как захватывающий детектив.

Глава 1. Основы синтаксиса языка и процесса разработки программ

1.1 Языки и синтаксис

Человек в процессе своего развития познает окружающий мир. Начинает общаться с людьми и животными, используя звуки. Эти звуки можно записать на листе бумаги, напечатать на печатной машинке или компьютере. При записи руководствуются определёнными правилами **- синтаксисом**.

Язык программирования можно рассматривать как человеческий язык, с использованием которого люди записывают **алгоритмы** (последовательность действий для решения задач). Как и у человеческого, у языка программирования имеются правила — **синтаксис**. Синтаксис в программировании очень строгий. Написанный текст использует ЭВМ, а машина не обладает интеллектом и не сможет обработать текст с ошибкой.

Если синтаксис – правила записи предложений (выражений), то правила записи одного слова – **лексика**. В программировании под лексикой можно понимать набор букв и символов, используемых при написании **выражений** (предложений).

Множество связанных выражений (предложений), состоящих из набора знаков (слов) представляют собой программу (текст).

Аналогия с человеческим языком	
Человеческий язык	**Язык программирования**
Буквы, знаки пунктуации	Буквы, символы
Слова	Операторы, переменные
Предложения	Выражения, команды
Текст	Программа
Воспринимается человеком	Выполняется компьютером
Возможны исключения из правил	Строгие правила

Возможны ошибки	Ошибки приводят к ошибкам
Сложно	Просто

Итак, **синтаксис языка программирования** — правила записи программ. Сам язык программирования, по сути является договорённостью между людьми, как записывать программы. Написание программ не требует специальной техники. Можно лежать на морском пляже, а программы записывать веточкой на мокром песке... До первого прилива...

1.2 Hello, World!

По хорошей давней традиции изучение любого языка начинается с вывода фразы `Hello, World!`. Под выводом, будем подразумевать вывод на экран, хотя место вывода можно изменить. Поздороваемся с миром с использование программы, написанной на языке `C++`:

```
#include <iostream>
int main() {
	std::cout << "Hello, World";
}
```

- Первая строка `#include <iostream>` означает подключение библиотеки `iostream`
- вторая строка `int main() {` означает начало программы
- третья строка `std::cout << "Hello, World";` означает команду вывода на экран
- четвёртая строка `}` означает конец программы

`std::` - пространство в котором содержатся стандартные команды. Команда `cout` одна из таких стандартных команд. Назначение у команд разное. Команда `cout` (от англ. `ConsoleOUT`) производит вывод на консоль (экран) текста, записанного после оператора `<<` (две треугольные скобки влево).

1.3 Функция main()

Рассмотрим представленный код более подробно, с точки зрения структуры программы.

```
int main() {// entry point

    // ... code here ...

/*exit point*/}
```

Любая программа на языке `C++` начинает и заканчивает своё выполнение в функции `main()`. Программа внутри функции реализует решение конкретной задачи, собственно, выполняет определённую функцию. Разберёмся, что такое функция и где у неё «внутри».

Можно провести аналогию с зелёной тетрадью. Сама тетрадь будет являться функцией. Конспект, который записывается в тетради — программа. Фигурные скобки — зелёные обложки. Подпись на обложке тетради — **сигнатура функции**:

1.3.1 Сигнатура функции

Сигнатура функции `main()` имеет следующий вид

```
int main()
```

Подробно рассматривать каждый символ не будем. Нужно запомнить, сперва пишется `int`, затем название `main`, после круглые скобки `()`. Отличие функции от других объектов, заключается именно в круглых скобках. По сигнатуре одну функцию можно отличить от другой.

1.3.2 Блок кода

Фигурные скобки служат для группировки нескольких выражений в единый программный блок. Они будут встречаться постоянно, поэтому важно усвоить данное понятие.

Блок кода (scope) – несколько выражений, сгруппированных вместе, которые должны выполняться последовательно друг за другом. В нашем случае блок кода принадлежит к функции `int main()`. Т.е. в фигурных скобках располагаются выражения, относящиеся к

функции `int main()`. Все программы, которые будут рассматриваться в ближайшее время, следует полностью располагать в этом `{ блоке кода }`.

По правилам оформления кода, каждое выражение внутри `{ }` должно начинаться с отступа:

```
оператор_1 {
    выражение_1;
    выражение_2;
}
оператор_2 {
    выражение_1;
    выражение_2;
    оператор_3 {
        выражение_5;
        выражение_6;
    }
}
```

1.3.3 Точка входа и выхода, комментарии

Содержимое, находящееся после двух слэшей `//` относится к комментариям. **Комментарии** не являются частью программы. Их не нужно писать. Комментарии содержат пояснения к сложным выражениям. Если требуется закомментировать часть строки или несколько строк, имеется возможность использования `/* многострочных комментариев */`.

В нашем примере комментариями обозначены точка входа и точка выхода из программы.

```
int main() {// entry point

    // ... code here ...

/*exit point*/}
```

`Entry point` — место, с которого программа начинает своё выполнение.

`Exit point` — место, в котором программа заканчивает своё выполнение.

1.3.4 Про пробелы, отступы и переносы строк

Пробелы, отступы и переносы строк, с точки зрения синтаксиса языка C++ не важны. Имеют значения только пробелы между словами. Например, между int и main() пробельные символы обязательны.

Программа "Hello, world" может быть записана и в две строки:

```
#include <iostream>
int main(){std::cout<<"Hello, world";}
```

#include <iostream> это инструкция для препроцессора, а не команда программы, поэтому писать её следует обязательно на отдельной строке.

И все же, принято запись каждого выражения начинать с новой строки. Внутри блока кода { } запись выражения начинается с отступа. Более того, слева и справа от оператора << ставится пробел. std::cout – единая команда, записывается без пробелов. Перед точкой запятой пробел тоже не ставят. А вот между) и { пробел желательно поставить.

Ниже представлен идеальный код, с правильно расставленными пробелами, отступами и переносами строк:

```
#include <iostream>


int main() {
	std::cout << "Hello, world";
}
```

1.4 Процесс разработки

Процесс разработки программы любой сложности состоит из следующих этапов:

- Продумывание алгоритма
- Написание кода
- Компиляция
- Исправление синтаксических ошибок

- Запуск
- Тестирование
- Исправление логических ошибок
- и так по кругу

1.4.1 Написание кода

Выше был подробно рассмотрен процесс написания программы `"Hello, World"`:

```
#include <iostream>

int main() {
	std::cout << "Hello, world";
}
```

Не ясным осталось одно, где писать? Лучше всего для первых программ подходит стандартный `Notepad (Блокнот)`. Нажмите `WIN+R` на клавиатуре введите `notepad` и нажмите `ENTER`. Откроется программа `Notepad`:

- Напечатайте в открывшемся `Notepad` код программы `"Hello, World"`.
- Создайте на диске `C:\` папку `prog`.
- Сохраните напечатанную программу в созданную папку `C:\prog`.
- При сохранение выберите `тип файла – все файлы`, в качестве названия файла укажите `hello.cpp`
- Получившийся файл `C:\prog\hello.cpp` является файлом с исходным кодом программы `source file`.

Source file – это текстовый файл, содержащий текст программы. Он обязан иметь расширение `.cpp`, для этого не забывайте при сохранении выбирать `тип файла – все файлы`.

1.4.2 Компиляция

Язык программирования C++ относится к **высокоуровневым**. Не будем подробно углубляться в классификацию языков. Слово высокоуровневый не означает, что он хороший и находится выше. Остальные языки (Java, C#, Perl, PHP, Python, Fortran, Lisp и т.д.) тоже являются высокоуровневыми.

Будем смотреть на словосочетание высокоуровневый язык, как *предназначенный для людей.*

Почему компьютер не способен выполнить программу, сохранённую в файле с исходным кодом `hello.cpp`? Сердцем компьютера, которое выполняет программу, является **микропроцессор**. Микропроцессор состоит из множества транзисторов - физических электронных устройств, которые могут принимать только два состояния: ток есть / тока нет. В логике принято обозначать эти состояния 1 / 0. Более подробно можете ознакомиться с работой микропроцессоров в книге FreeASM того же самого автора, что и эта книга.

Таким образом, компьютер способен выполнить программу, записанную только как последовательность из двух состояний:

- тока нет (0)
- ток есть (1)

Программа "Hello, World" состоит и высокоуровневых слов. Чтобы компьютер смог её выполнить, требуется перевести код, написанный на высокоуровневом языке C++ в низкоуровневый код – **машинный язык**.

Компиляция – процесс перевода высокоуровневого кода в низкоуровневый.

Компилятор – программа, которая анализирует исходных код, проверяет его на наличие ошибок, в случае их отсутствия производит перевод высокоуровневого исходного кода в машинный код.

Процесс компиляции в простейшем случае можно представить следующей схемой:

Результатом работы компилятора является скомпилированная программа – исполняемый файл. В операционной системе `Windows` исполняемые файлы имеют расширение `.exe`.

Именно файл `hello.exe` требуется запустить, чтобы увидеть результат работы программы.

Не вдаваясь в подробности, низкоуровневый машинный код можно назвать `Assembler`.

Компилирование и запуск программы

Как же получить из `hello.cpp` готовую программу `hello.exe`? Потребуется установить компилятор `g++`, который идёт в комплекте со множеством других компиляторов:

http://www.equation.com/servlet/equation.cmd?fa=fortran

Выбирайте 64-битную версию программы. Перед установкой создайте папку `compilators` на диске диске `C:\`. При установке укажите путь именно к созданной папке `C:\compilators`. После установки обязательно перезагрузите компьютер.

Компилятор не имеет графической оболочки. Придётся работать с командной строкой. Откройте командную строку, для этого нажмите сочетание клавиш `WIN+R => cmd => ENTER`. Командная строка позволяет перемещаться по файловой системе и запускать программы так, как будто вы это делаете с использованием программы проводник (мой компьютер).

В открывшемся окне консоли, слева от мигающего курсора располагается приглашение командной строки:

```
c:\users\admin> _
```

Приглашение указывает на текущее местоположение в файловой системе. Чтобы увидеть содержимое текущей папки выполните команду `dir`. Просто напечатайте `dir` и нажмите `ENTER`.

Для перехода в другую директорию (папку) используется команда `cd`. Выполните команду:

```
cd C:\prog
```

чтобы перейти в директорию, в которой находится source file `hello.cpp`.

Выполните команду `dir`, чтобы удостовериться, что файл `hello.cpp` присутствует в выбранной директории.

Для запуска компилятора выполните команду `g++`:

```
g++
```

Если появилась ошибка следующего содержания, значит компилятор установлен верно:

```
g++: fatal error: no input files
compilation terminated.
```

Ошибка говорит о том, что компилятору на вход не был передан файл с исходным кодом программы. Если появилась ошибка операционной системы, типа:

```
«g++ не является программой или исполняемым файлом»
```

следует добавить путь в системную переменную `PATH` до директории, в которой хранится компилятор `g++.exe`. Подробную инструкцию не составит найти труда в интернете.

Считаем, что компилятор успешно запускается из командной строки. Добавим в команду компиляции пару параметров – аргументов:

```
g++.exe hello.cpp -o hello.exe
```

в качестве первого аргумента указано название файла с исходным кодом, который следует скомпилировать, после флага `-o` указывается имя файла, в который будет помещён результат компиляции. После нажатия `ENTER` начнётся компиляция, которая может занять несколько

секунд. В случае успешной компиляции в консоль ничего выведено не будет. Просто отобразиться новое приглашение:

```
C:\prog> g++.exe hello.cpp -o hello.exe
C:\prog>_
```

Выполните команду `dir` и обратите внимание, что в папке `c:\prog` появился новый файл `hello.exe` – первая программа, которую теперь можно запустить. Для запуска, просто введите название файла и нажмите `ENTER`.

```
hello.exe
```

Миру приятно, что нам удалось с ним поздороваться.

Указывать имя выходного файла `-o hello.cpp`, в команде компиляции не обязательно. Результат в случае успешной компиляции по умолчанию записывается в файл с именем `a.exe`. Более того расширение `.exe` можно не указывать. Теперь команда компиляции и запуска будут выглядеть следующим образом:

```
C:\prog> g hello.cpp
C:\prog> a
```

Консоль

Ниже представлен перечень команд командной строки, которые позволят расширить кругозор:

1. `dir` или `ls` – отобразить содержимое текущей директории
2. `notepad` – открыть блокнот
3. `notepad hello.cpp` – открыть в блокноте файл hello.cpp из текущей директории
4. `cd <directory>` – change directory

   `cd c:\` – перейти в корень диска `c:\`

   `cd c:\users` – перейти в директорию `users` на диске `c:\`

   `cd my_folder` – перейти из текущей директории в директорию `my_folder`

`cd ..` – вернуться назад на один уровень выше

`cd ..\..` – вернуться назад на два уровня выше

5. `md <folder>` или `mkdir <folder>` – make directory folder

   `md my_folder` – создать директорию `my_folder` в текущей директории

6. `copy <file1> <file2>` – copy `file1` to `file2`

   `copy main.cpp main2.cpp` – копирует файл `main.cpp` из текущей директории в файл `main2.cpp` в текущей директории

   `copy main.cpp c:\my_prog\` – копирует файл `main.cpp` из текущей директории в директорию `c:\my_prog\`

   `copy c:\my_prog\main.cpp` – копирует файл main.cpp из директории c:\my_prog в текущую директорию.

7. `move <file> <dir>` – move `file` in directory `dir`

8. `del main.cpp` – delete file `main.cpp` from current directory

Полезные утилиты командной строки:

1. `ipconfig` – узнать `ip` адрес компьютера
2. `ping ya.ru` – проверить соединение с `ya.ru`
3. `tracert ya.ru` – посмотреть маршрут по которому проходит запрос от вашего компьютера до `ya.ru`
4. `where notepad` – показать путь, по которому расположена программа `notepad`
5. `shutdown /?` – выключение компьютера, отобразить справку

   `shutdown /s /t 300` выключить компьютера через 5 минут

Можно создать ярлык, например, для выключения компьютера. Для этого кликните на рабочем столе ПКМ и выберите создать ярлык.

В качестве расположения объекта укажите `shutdown -s -t 10`.

1.4.3 Исправление ошибок

Откроем консоль `WIN+R => cmd => ENTER`

- Перейдем в директорию `prog`:

```
cd c:\prog
```

- Скопируем файл `hello.cpp` в файл `my_err.cpp`

```
copy hello.cpp my_err.cpp
```

- Откроем файл в `Notepad`:

```
notepad my_err.cpp
```

Напишем следующий код:

```
#include <iostream>

int main(){
	std::cout << "Hello, World" << std::endl
	std::cout << "Hi";
}
```

В коде намеренно допущена ошибка. Скомпилируем программу (обязательно сохраняйте файл `.cpp` перед компиляцией, компилируется содержимое файла, которое сохранено в нем на данный момент):

```
g++ my_err.cpp
```

Компилятор в результате анализа нашего исходного кода обнаружит ошибку и выведет информацию о ней:

```
my_err.cpp: In function ‘int main()’:
my_err.cpp:3:49: error: expected ‘;’ before ‘std’
   3 |       std::cout << "Hello, World" << std::endl
     |                                                 ^
     |                                                 ;
   4 |       std::cout << "Hi";
     |       ~~~
```

Компилятор подробно описывает текст ошибки и указывает место где её необходимо исправить. Воспользуйтесь словарём и переведите описание ошибки:

```
error: expected ‘;’ before ‘std’
```

Давайте исправим ошибку, в Notepad добавим точку с запятой:

```
#include <iostream>

int main(){
    std::cout << "Hello, World!" << std::endl;
    std::cout << "Hi";
}
```

Сохраним файл, и скомпилируем вновь:

```
g++ my_err.cpp
```

Ошибок нет. Запустим программу, для этого введите в консоли `a.exe` или просто `a`. Приятно, что мир поздоровался с нами, да ещё с новой строки!

1.4.4 Тестирование

Выполните следующие действия самостоятельно:

- Скопируйте файл `my_err.cpp` в файл `logic_err.cpp`.
- Откройте его в Notepad.
- Замените Hi на Snow.
- Скомпилируйте и запустите программу.

Программа скомпилировалось и запустилась. Однако, вывод какой-то странный. На экране появилось не то, что ожидалось. Предполагалось, что мир должен ответить взаимностью. Если программа работает не так как ей следовало бы, имеется **логическая ошибка**.

Логические ошибки обнаруживаются в процессе **тестирования программы**. Тестирование заключается в многократном запуске программы с различным набором входных данных. Исправлять и находить логические ошибки труднее, поскольку компилятор их не может

обнаружить. При этом, для некоторых входных данных программа может выдавать правильный результат, а для некоторых не правильный. Особо часто логические ошибки встречаются при граничных значениях (очень больших или очень малых) допустимых входных данных.

Существует подход к разработке программ через тестирование – сперва пишутся тесты. В процессе разработки программа постоянно тестируется на данных тестах. Логические ошибки, таким образом, удаётся свести к минимуму.

Исправьте `snow` на `Hello, my friend`. Скомпилируйте и запустите программу. Убедитесь, что удалось исправить логическую ошибку.

Глава 2. Типы и переменные

Напишем программу, которая выводит сумму двух чисел на экран.

```
#include <iostream>


int main(){
	std::cout << 2 + 3 << std::endl; // 5
}
```

Математическое выражение `2 + 3` записано без двойных кавычек. В данном случае компилятор сгенерирует такой машинный код, который будет складывать два числа `2` и `3` и выводить результат `5` на экран. Если записать `2 + 3` в двойных кавычках, в результате будет просто выведено пять символов: `2, пробел, +, пробел, 3`

```
std::cout << "2 + 3" << std::endl; // 2 + 3
```

Следующие две строки:

```
#include <iostream>
int main() {
```

в примерах кода записываться больше не будут. Разумеется они обязательны. Подразумевается, что все выражения записываются в `{ блоке кода }`, относящемуся к функции `int main()`.

2.1 Объявление переменной

Программы предназначены для обработки информации. **Переменная** - место хранения информации, обрабатываемой программой.

Попробуем сделать следующее. Значения `2` и `3` поместить в переменные. На экран выводить результат сложения значений, хранимых в переменных.

Переменную необходимо создать, прежде чем в неё поместить значение.

Объявление переменной – выражение, создающее переменную. При объявлении переменной указывается **тип** и **имя**.

Тип переменной – указывает какого рода информацию в переменной можно хранить.

Пока будем работать только с целыми числами. Тип для целых чисел называется `int` (от англ. `integer`).

Имя переменной – позволяет из программы обращаться к значению, которое хранится в переменной. К имени предъявляются следующие требования:

- Может содержать только символы из набора `[a..Z0..9_]`, т.е. латинские буквы, цифры и знак подчёркивания.
- Должно начинаться с буквы или знака подчёркивания.
- Не может содержать пробелов.
- Регистр имеет значение: `Num`, `num`, `NuM` – три разных имени.
- Внутри одного `{ блока кода }` имена не могут повторяться.
- Должно быть осмысленным.

Создадим две переменных с именами `num_1` и `num_2`, которые будут хранить целые числа (информацию типа `int`). Запишем в них значения `2` и `3`. Выведем результат сложения значений, хранимых в переменных, на экран:

```
int num_1; // объявление переменной с именем num_1 типа int
int num_2; // объявление переменной с именем num_2 типа int
num_1 = 2; // запись значения 2 в переменную с именем num_1
num_2 = 3; // запись значения 3 в переменную с именем num_2
std::cout << num_1 + num_2; // 5
```

На переменные можно смотреть как на контейнеры для хранения информации. У каждого такого контейнера имеется уникальное имя. Чтобы создать контейнер с именем `math_weight`, необходимо добавить в программу строку с выражением:

```
int math_weight;
```

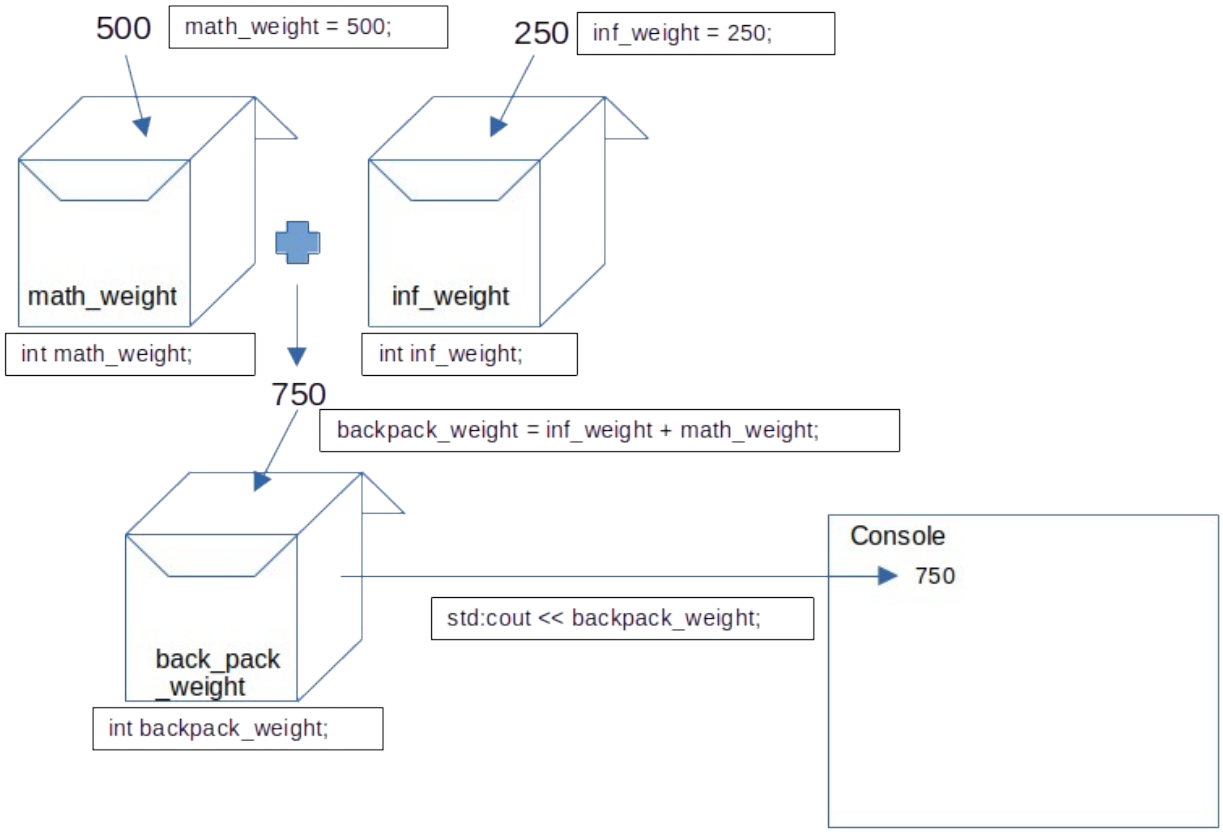
После того как контейнер создан, в него можно положить число, указав выражение:

```
math_weight = 500;
```

Рассмотрим программу, представленную ниже.

```
int math_weight;
int inf_weight;
int backpack_weight;
math_weight = 500;
inf_weight = 250;
backpack_weight = math_weight + inf_weight;
std::cout << "backpack_weight: " << backpack_weight << std::endl;
```

Создаётся три контейнера. В математику и информатику заносятся соответствующие веса учебников. Вычисляется сумма весов и результат помещается в контейнер вес рюкзака `backpack_weight`. На экран выводится строка (текст) `"backpack_weight: "`, значение, хранимое в контейнере `backpack_weight`, и перенос строки.



Согласитесь, программный код выглядит более лаконично, чем графическая схема.

2.2 Запуск и выполнение программы.

Где же на самом деле хранятся значения переменных? Это место располагается в оперативной памяти и называется **стеком вызовов**. При запуске уже скомпилированной `.exe` программы операционная система совершает примерно следующие действия:

- Выделяет место в оперативной памяти – запускает процесс. Это тот самый процесс, информацию о котором видно в `диспетчере задач Windows`.
- Копирует `"машинный код"` из `.exe` файла в процесс (оперативную память).
- Выделяет в процессе место – **стек вызовов**, для хранения значений переменных.
- Вызывает функцию `main()` - отправляет на выполнение микропроцессору первую команду, относящуюся к функции `main()`, из машинного кода, получившегося в результате компилирования высокоуровневого кода
- После выполнения последней команды из функции `main()` уничтожает процесс.

2.2.1 Схема оперативной памяти

Оперативную память можно представить в виде набора ячеек, каждая весом по 1 байт. У каждой ячейки имеется адрес, который принято записывать в 16-й системе счисления. Шестнадцатеричная система используется, поскольку число, записанное в двоичной системе тяжело для восприятия человеком. В привычную 10ю систему счисления из 2й переводить долго. А вот из 2й в 16ю и обратно перевод осуществляется легко, при определённом навыке его можно сделать в уме.

Память размером 1 Кбайт можно представить следующей таблицей:

Память объёмом 1 Кбайт	
Адрес	**Содержимое**
0x000	0b00000000
0x001	0b00101011
0x002	0b11001101
	...
0x3FD	0b11100101

0x3FE	0b11110011
0x3FF	0b11110101

В первом столбце указан адрес ячейки в 16-м формате. Префикс `0x` – означает, что число, расположенное после `0x`, записано в 16-й системе счисления. Для 10-й можно использовать префикс `0d`, для двоичной `0b`, для восьмеричной `0o`.

Посмотрим на адрес последней ячейки `0x3FF = 0b1111111111 = 1023`. Он равен `1023`, а не `1024`. Это связано с тем, что адрес первой ячейки равен нулю, таким образом, всего получается `1024` ячейки. Каждая ячейка весит один байт, следовательно общий объем памяти 1 Кбайт. Размер 4 Гб, обозначает чуть более четырёх миллиардов таких ячеек.

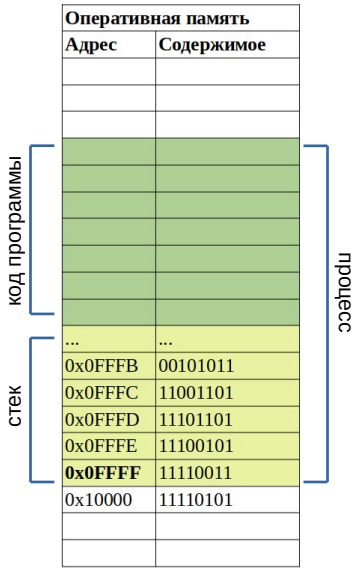
Каждая ячейка занимает объем 1 байт, что равно 8 бит. Известно, что **бит** – наименьшая единица измерения информации, которая может хранить либо `0`, либо `1`. Таким образом, каждый байт хранит восемь двоичных разрядов. Запишем диапазон целых неотрицательных чисел, которые можно сохранить в одном байте:

Диапазон целых неотрицательных чисел		
Система счисления	**Минимальное**	**Максимальное**
binary	0b00000000	0b11111111
decimal	000	255
hexadecimal	0x00	0xFF

Эти числа часто встречаются при работе с информационными технологиями.

2.2.2 Процесс

Представим, что мы запустили программу. Операционная система выделила процесс – место в оперативной памяти. В ячейки, обозначенные зелёным цветом, скопировала машинный код скомпилированной программы. После кода, определила место для нескольких сотен тысяч ячеек, в которых будет располагаться стек вызовов. Стек вызовов обозначен жёлтым цветом. Белым цветом обозначены ячейки оперативной памяти, не задействованные в процессе. В этих ячейках могут быть запущены другие процессы, в том числе, располагаться машинный программный код самой операционный системы.



2.2.3 Выполнение программы

Скомпилированная программа представляет собой набор машинных команд. Каждая машинная команда представляет собой целое число. Все команды (числа) после запуска программы хранятся в оперативной памяти. Микропроцессор выполняет данные команды (числа) по порядку, начиная с первой команды после `int main() { entry point`. Вплоть до выполнения последней машинной команды, которая в исходном коде располагалась непосредственно перед `exit point }`. После чего процесс завершается.

Если при выполнении программы не было ошибок, то процесс возвращает операционной системе число `0`. При возникновении ошибки или нештатной ситуации (например, деление на ноль) процесс завершается аварийно, а операционной системе возвращается `код ошибки`. Операционная система решает, что с этим делать.

Можно разработать программу, которая при успешной компиляции будет сразу запускать результирующий скомпилированный файл. Компилятор, так же как и ваша программа, возвращает `0` в случае успешной компиляции. Таким образом, можно отправить файл на компиляцию и если компиляция прошла успешно, произвести запуск программы, иначе вывести в окно с ошибками информацию об ошибке, которую вернул компилятор. Примерно так поступают сложные интегрированные среды разработки типа `Visual Studio`, когда вы нажимаете на кнопку скомпилировать и выполнить (`Build and Run`) или `F5` на клавиатуре.

2.3 Основные операторы

Операторы это знаки, которые позволяют совершать операции над располагающимися вокруг них объектами. Эти объекты называют **операндами**.

Рассмотрим выражение:

```
2 + 3
```

Здесь `+` является оператором, а `2` и `3` операндами. Оператор `+` в данном примере осуществляет арифметическое (бывают и другие виды) сложение операндов.

В выражении `a * b`, происходит умножение значений, хранимых в `a` и `b`.

А вот ещё, не столь очевидный пример:

```
std::cout << "Hello";
```

Символ (точнее два символа) `<<` тоже является оператором. Он также имеет два операнда std::`cout` и `"Hello"`. Оператор `<<` выводит значение, хранящееся в правом операнде в поток, расположенный в левом операнде. `std::cout` – это поток вывода на консоль. Если создать файловый поток `file` и написать нечто вроде

```
file << "Hello";
```

вывод будет осуществляться в файл.

Операторы, имеющие два операнда, называются бинарными. Примеры бинарных операторов:

```
+, *, -, /, %, =, ==, !=, &&, ||, <<
```

Унарные операторы, действуют на один операнд: `i++, !isExist`

Тернарные операторы, производят операцию над тремя операндами:

```
cond ? true_value : false_value
```

2.3.1 Присваивание

Камнем преткновения при изучении программирования часто служит оператор присваивания. Это связано с тем, что он обозначается знаком `=`, который очень похож на знак равенства из математики. Запомните, `=` в языке программирования `C++` это не равно, это оператор присваивания.

Оператор присваивание – записывает (копирует) значение из правого операнда в левый.

Данный оператор широко использовался в примерах выше:

```
math_weight = 125;
```

Значение из правого операнда `125` записывается в левый операнд – переменную `math_weight`

Более сложный пример:

```
backpack_weight = math_weight + inf_weight;
```

Значение из правого операнда (`math_weight + inf_weight`) записывается в левый операнд – переменную `backpack_weight`. Правый операнд в данном выражении сложный и сам состоит из оператора `+` и двух операндов. В таких выражениях, сперва вычисляется значение правого операнда. После чего происходит присваивание вычисленного значения левому операнду.

Что выведет следующая программ

```
int a, b;
a = 5;
b = 6;
b = a;
std::cout << a << b;
```

Верный ответ `55`.

- во второй строке значение `5` присваивается переменной `a`
- в третьей строке переменной `b` присваивается значение `6`
- в четвёртой строке переменной `b` присваивается значение переменной `a` (в переменную `b` записывается значение из переменной `a`)

Таким образом, и в a и в b хранится значение 5. Другими словами ("бэ равняется a, но не наоборот!"). Если сложно понять, смотрите на оператор =, как на стрелку влево b←a, что логично обозначает запись из a в b.

Ещё более странный пример:

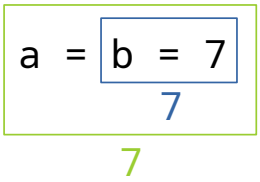
```
int a = 5;
a = a + 2;
std::cout << a;
```

Результатом будет значение 7. Сперва оператор + совершает действия над операндами a и 5. Получается значение 7. Далее это значение записывается в переменную a.

Следует отметить, что результатом присваивания является само присваиваемое значение. Поэтому такая форма записи допустима:

```
int a, b;
a = b = 7;
```

Выражение b = 7 даст 7. Полученная таким образом семёрка присваивается переменной a:



Данная особенность может пригодиться при использовании оператора присваивания = в выражениях, выступающих в качестве условий ветвлений и циклов, которые будут рассмотрены в следующих разделах.

2.3.2 Арифметические операторы

Здесь все относительно просто, но есть нюансы:

`a + b` сложение

`a – b` вычитание

`a * b` умножение

`a / b` деление

`a % b` остаток от деления

Если оператор деления применяется к операндам, представляющим собой целые числа, то дробная часть отбрасывается. Происходит именно отсечение дробной части, а не округление:

```
int b = 128 / 10;
std::cout << b; // = 12, а не 13 (при округлении 12.8 = 13)
```

Для целых чисел также предусмотрена операция вычисления остатка от деления `%`:

```
int b = 128 % 10;
std::cout << b; // = 8, т.к. 128 разделить на 10 равно 12 целых и 8 в остатке
```

Помимо бинарных операторов имеется ряд унарных операторов:

`-a` меняет знак числа на противоположный

`++i` инкремент – увеличение на единицу

`--i` декремент – уменьшение на единицу

Если требуется совершить действие над одной переменной и результат оставить в ней же, можно воспользоваться одним из **операторов составного присваивания**:

```
+=, -=, *=, /=, %=
```

Таким образом, инкремент можно записать четырьмя различными способами

```
i = i + 1
i += 1
i++
++i
```

Обратите внимание, что инкремент бывает префиксный `++i` и постфиксный `i++`. Отличие в порядке выполнения инкремента.

Для префиксного, сперва прибавляется 1, затем вычисляется оставшееся выражение уже с увеличенным значением.

Для постфиксного сперва вычисляется выражение со старым значением, затем происходит увеличение на 1.

Следующие примеры помогут понять различие.

Постфиксный:

```
int a = 2, b = 5, c = 0;
c = a + b++;
std::cout << c << "," << b; // 7, 6
```

Префиксный:

```
int a = 2, b = 5, c = 0;
c = a + ++b;
std::cout << c << ", " << b; // 8, 6
```

Обратите внимание, насколько короче стал код. Синтаксис языка `C++` позволяет объявлять сразу несколько переменных через запятую. Переносы строк и пробелы по прежнему не имеют значение, главное правильно расставить знаки препинания:

```
int     a = 2,
        b = 5,
        c = 0;
c = a + ++b;
std::cout << c << ", " << b; // 8, 6
```

Полная свобода для творчества, ещё одно неоспоримое преимущество языка `C++`.

Существует множество других операций, такие как сдвиги, побитовые операции. По мере необходимости, некоторые из них будут разобраны более подробно.

2.3.4 Операторы ввода / вывода

Оператор вывода (см. раздел *перегрузка операторов* в главе 9) << позволяет вывести правый операнд в поток вывода из левого операнда:

```
std::cout << "backpack_weight: ";
```

Можно выводить несколько объектов, разделяя их оператором <<:

```
std::cout << "average grade: " << av_grade << std::endl << "\tnew line";
```

Объекты отправляются в поток последовательно слева направо. Для перевода строки в поток необходимо отправить команду `std::endl`.

Специальный символ `"\t"` позволяет добавить в текст отступ, как будто в этом месте была нажата клавиша `TAB`. Спецсимвол `"\t"` удобно использовать для придания выводимой информации структурированного вида. Например, вывод информации в виде столбцов.

Оператор ввода направлен в противоположную сторону >> и работает с потоком ввода `std::cin`. По умолчанию, поток ввода связан с клавиатурой:

```
int a;
std::cin >> a;
a *= a;
std::cout << a;
```

Если скомпилировать данный код и запустить, полученный исполняемый файл, то создастся ощущение, что программа зависла. В консоли будет одиноко мигать курсор. На самом деле программа ждёт, пока пользователь введёт число. Введите число `5` и нажмите `ENTER`.

Также как и с выводом, можно заставить пользователя вводить сразу несколько значений:

```
int a, b, c;
std::cin >> a >> b >> c;
std::cout << (a + b + c) / 3.0;
```

Представленная программа ожидает ввода трёх целых чисел. После чего выводит среднее арифметическое. Числа можно ввести сразу три через пробел, либо по одному, каждый раз нажимая `ENTER`. Введите `1 2 5` и нажмите `ENTER`.

Обратите внимание, что деление осуществляется на дробное число `3.0`, а не на целое `3`. Иначе дробная часть отсечётся. Если требуется увеличить количество выводимых знаков после запятой, добавьте команду `std::cout.precision(17)` перед выводом :

```
int a, b, c;
std::cout.precision(17);
std::cin >> a >> b >> c;
std::cout << (a + b + c) / 3.0;
```

Точность вычислений при этом не изменяется. Просто выводится бо́льшее количество знаков. Кстати, тип дробных чисел в `C++` называется `double`.

2.3.5 Прочие операторы

Помимо арифметических бывают операторы:

- сравнения `>, <, >=, <=, !, ==, !=`
- логические `!, &&, ||`
- побитовые `~, &, |, ^, <<, >>`
- для работы с указателями и членами класса `a[b], *a, &a, a->b, a.b, a→*b, a.*b`

А ещё:

- функтор `a(a1, a2)`
- оператор запятая `a, b`
- тернарная условная операция `a ? b : c`
- оператор расширения области видимости `a::b` (именно он встречается в `std::cout` и `std::endl`)
- вычисление размера `sizeof(a)`
- выделение памяти `new type`
- выделение памяти для массива `new type[n]`
- освобождение памяти `delete a`
- освобождение памяти, занятой массивом `delete[] a`

2.3.6 Приоритет

Приоритет определяет в каком порядке операции выполняются в сложном выражении. В ближайшее время будет интересовать приоритет выполнения арифметических операций. Он такой же как в математике: `()` `*` `/` `%` `+` `-`

В сложных выражениях порядок выполнения операций следующий:

1. Выражение в скобках
2. Арифметические операции
3. Операции сравнения
4. Логические операции

Глава 3. Следование

Алгоритм – последовательность действий для решения поставленной задачи. С использованием трёх алгоритмических конструкций можно реализовать любой алгоритм. Перечислим эти конструкции:

- следование
- ветвления
- циклы

Простейшая и самая главная конструкция **следование**. Она явным образом входит в состав двух остальных.

Следование – алгоритмическая конструкция, в которой команды выполняются последовательно друг за другом.

К слову, микропроцессор может выполнять только последовательный набор команд. Просто некоторые команды настолько хитрые, что провоцируют скачки из одной части программы в другую и обратно, тем самым реализуя ветвления и циклы.

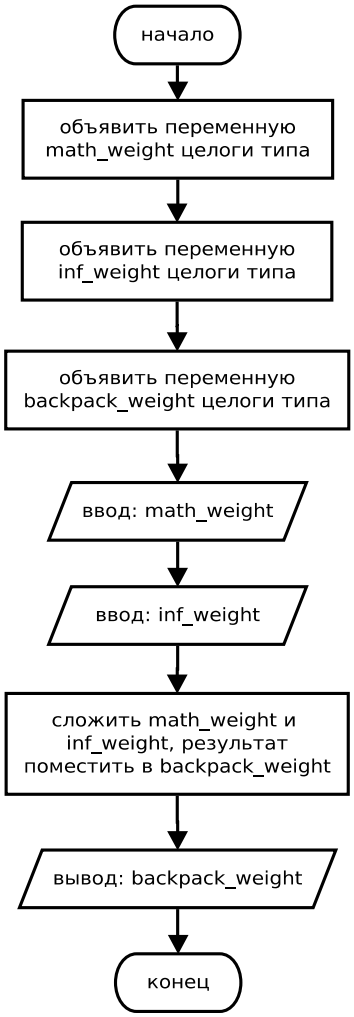
3.1 Блок схема

Алгоритмические конструкции удобно представлять с использованием блок схем. Команды или выражения на блок схемах изображаются прямоугольниками. Команды ввода / вывода параллелограммами, условия и циклы ромбами, начало и конец, скруглёнными прямоугольниками.

Рассмотрим программу "Калькулятор веса рюкзака учащегося":

```
int math_weight;
int inf_weight;
int backpack_weight;
std::cin >> math_weight;
std::cin >> inf_weight;
backpack_weight = math_weight + inf_weight;
std::cout << "backpack_weight: " << backpack_weight << std::endl;
```

Составим по ней блок-схему алгоритма:



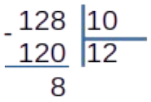
Из схемы видно, что одна команда выполняется вслед за другой по стрелочкам. Данная схема, отражает алгоритмическую конструкцию следование.

3.2 Сумма цифр числа

Разберем алгоритм для следующей задачи: "Найдите сумму цифр трёхзначного числа, введённого пользователем". На основе данного алгоритма изучим два важных **паттерна** (шаблона):

1. разбор числа по цифрам
2. «накопитель»

Разбор числа по цифрам основан на свойствах систем счисления. Поделим число 128 нацело на 10.



Целая часть получится равной 12, а остаток равен 8.

Вывод:

- Остаток от деления числа на основание системы счисления равен последней цифре числа в данной системе счисления.
- Целая часть от деления на основание системы счисления отсекает последнюю цифру от числа в данной системе счисления.

В синтаксисе C++ эти утверждения выглядят следующим образом:

```
128 % 10 = 8
128 / 10 = 12
```

Последовательно применяя операции % и / можно перебрать все цифры числа.

"Паттерн разбора числа по цифрам" выглядит следующим образом:

1. Извлечь последнюю цифру из числа N % 10
2. Отсечь последнюю цифру от числа N = N / 10 или N /= 10
3. Извлечь последнюю цифру из числа N % 10

4. Отсечь последнюю цифру от числа `N = N / 10 или N /= 10`
5. Извлечь последнюю цифру из числа `N % 10`, либо просто `N`, т. к. после отсечения двух цифр на предыдущих шагах алгоритма в `N` осталась одна цифра.

Чтобы вычислить сумму всех цифр следует реализовать паттерн `"накопитель"`:

1. Объявить переменную `sum`.
2. Записать в неё `0`, т. к. сумма цифр не известна.
3. Прибавлять по одной цифре к сумме, используя выражение:

   ```
   sum = sum + последняя цифра
   ```

   либо оператор составного присваивания

   ```
   sum += последняя цифра
   ```

Объединим эти два паттерна в единый алгоритм и запишем его в словесной форме:

1. Объявить переменную `число`.
2. Объявить переменную `сумма`.
3. Ввод в переменную `число` значения из клавиатуры.
4. Задать `сумме` значение `ноль`
5. Прибавить к `сумме` последнюю цифру: `сумма += число % 10`
6. Отсечь от `числа` последнюю цифру: `число /= 10`
7. Прибавить к сумме последнюю цифру...
8. Отсечь от `числа` последнюю цифру . . .
9. Прибавить к `сумме` последнюю цифру . . .
10. Вывод `суммы` на экран

Алгоритм можно записать и в виде псевдокода, на каком-нибудь выдуманном языке программирования My++:

```
начало
    цел число
    цел сумм
    ввод (число)
    сумм := 0
    сумм := сумм + число mod 10
    число := число div 10
    сумм := сумм + число mod 10
    число := число div 10
    сумм := число
    вывод (число)
конец
```

В данном языке вместо `/` используется оператор `div`, вместо `%` оператор `mod`, а вместо `=` оператор `:=`

Теперь вы в состоянии написать программу на языке программирования C++. Воспользуйтесь таблицей перевода команд и операторов из языка My++ в язык C++.

C++	**My++**
`int a;`	`цел a`
`std::cin >> a;`	`ввод (a)`
`std::cout << a;`	`вывод (a)`
`=`	`:=`
`/`	`div`
`%`	`mod`

3.3 Swap

Решите самостоятельно следующую задачу:

> Пользователь вводит два числа. Первое число присваивается одной переменной, а второе другой. Необходимо поменять значения переменных местами.

Переформулируем:

> Имеется два стакана. Следует поменять содержимое стаканов местами.

Разумеется требуется третий стакан для временного использования. Его обычно так и называют `temp` (от английского `temporary` – временный).

Если трудно написать код сразу, возьмите три стакана. Подпишите их разными именами. Налейте в один стакан чай, в другой воду. Сделайте так, чтобы в стакане с водой оказался чай, а в стакане с чаем вода. Каждое переливанию в коде соответствует выражение `стакан_1 = стакан_2`, при этом переливание происходит из `стакан_2` в `стакан_1`. Попробуйте составить блок-схему, написать псевдокод на языке `My++`.

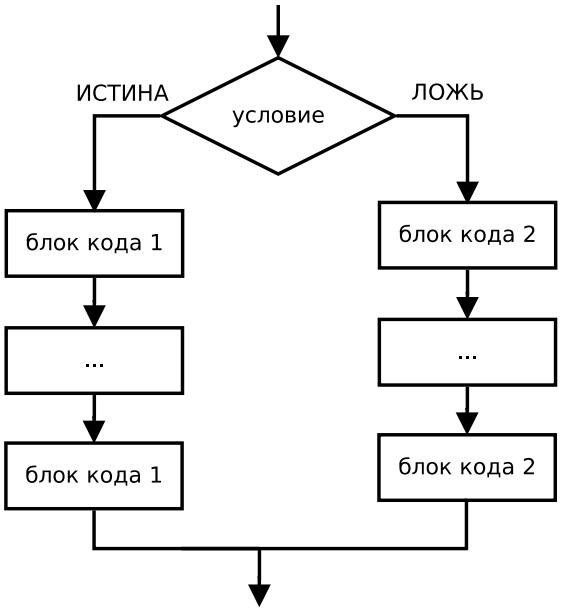
Обмен местами значений переменных может использоваться при сортировке массива. Если б*о*льший элемент стоит левее м*е*ньшего, то их следует поменять местами.

Глава 4. Ветвление

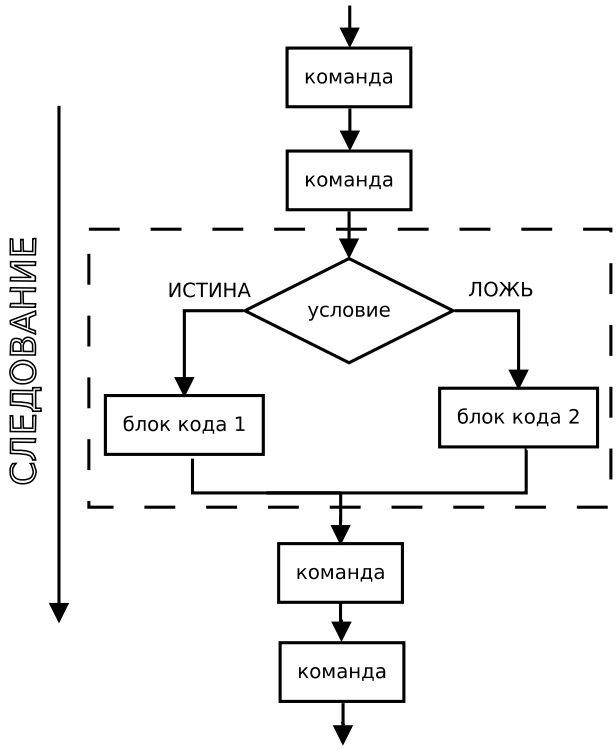
Выбор — неотъемлемое право и обязанность каждого человека. Просыпаемся, идём открываем холодильник и делаем выбор. Смотрим за окно на погоду и делаем выбор. Доходим до развилки в лесу и делаем выбор. Если в алгоритме делается выбор, следует использовать ветвление.

Ветвление — алгоритмическая конструкция, позволяющая выполнить команды из `{блока кода 1}` в том случае, если некоторое условие выполняется, и команды из другого `{блока кода 2}` в противном случае.

На блок схеме условие в ветвлении обозначается ромбом, от которого отходят две стрелки. По первой стрелке выполнение программы пойдёт, если условие `истинно`, по другой стрелке, если условие `ложно`, т. е. в противном случае:



Ветвление, находящееся посреди других команд программы, можно рассматривать как одну большую команду в составе следования:



Под словами ИСТИНА и ЛОЖЬ следует понимать:

ИСТИНА	ЛОЖЬ
да	нет
правда	не правда
выполняется	не выполняется
Число 1 и любое другое, отличное от 0	Число 0
true	false

4.1 Конструкция if else

В синтаксисе языка C++ ветвление реализуется через конструкцию `if else`:

```
// ... before code
if(/*condition*/) {
	// ...code_block_1
}
else {
	// ...code_block_2
}
// after_code...
```

Условие `condition` записывается в круглых скобках после ключевого слова `if`. Если значение `condition` оказывается истинным (т. е. не нулём), выполняются команды из `{ code_block_1 }`, после чего выполнение переходит к выражениям `after_code`. Блок кода `{ code_block_2 }` пропускается.

Если значение `condition` оказывается ложным (т. е. нулём), блок кода `{ code_block_1 }` пропускается, выполняются выражения из `{ code_block_2 }`. Далее выполнение переходит к выражениям `after_code` после ветвления.

Следует отметить, что у блока `else` нет круглых скобок, т. к. у него нет условия. Этот блок выполнится, если условие `if()` ложно. Если согласно алгоритму, в случае ложного условия никаких действий происходить не должно, блок `else` можно не указывать.

Рассмотрим простейший пример программы `"Что больше": Пользователь вводит два числа, вывести информацию о том, какое из чисел больше, первое или второе`

```
int first, second;
std::cin >> first >> second;
if(first > second) {
	std::cout << "first more";
}
else {
	std::cout << "second more";
}
```

4.2 Множественный выбор

Протестируем разработанную программу. Составим набор тестов. Тестирование получится эффективным, если каждый тест будет принципиально отличаться от остальных.

Например, можно составить следующий набор тестов:

first	**second**	**out_real**	**out_ideal**
5	8	Second more	Second more
8	5	First more	First more
-10	-4	Second more	Second more
10	0	First more	First more
5	5	**Second more**	equal

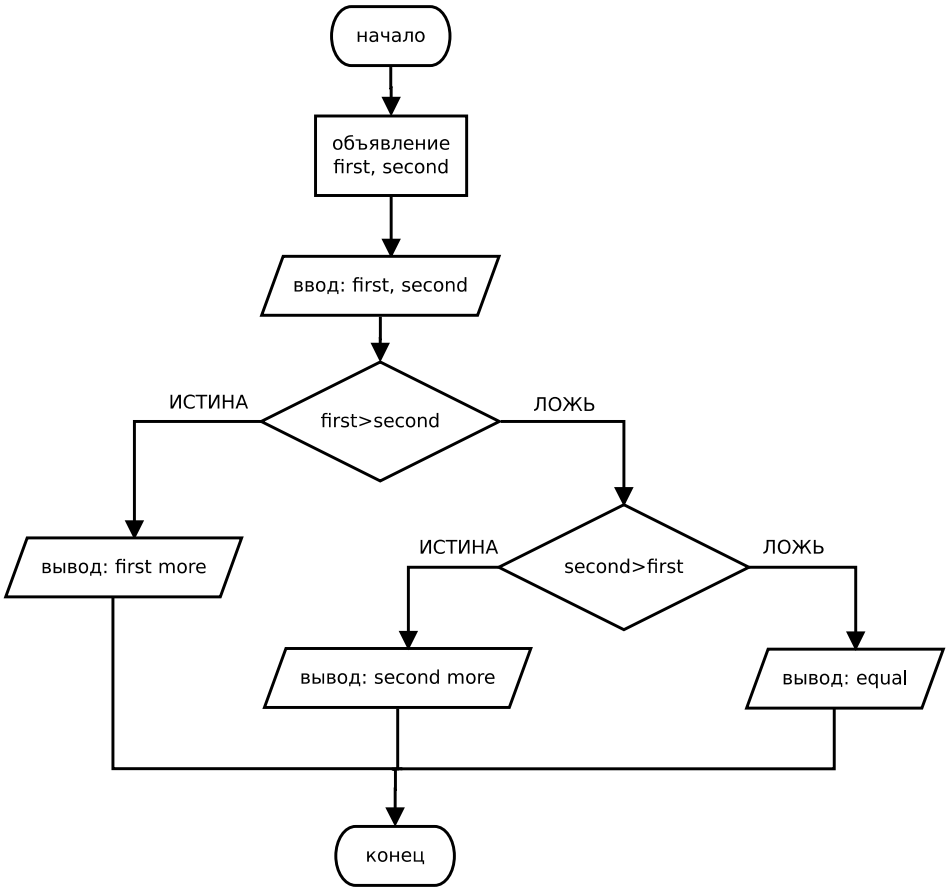
Видим, что на последнем тесте `5, 5` программа даёт сбой, выводит `second more`, хотя числа равны. Таким образом, тестирование позволило выявить логическую ошибку.

Вложенное в блок `else {}` ветвление позволит преодолеть логическую ошибку:

```
int first, second;
std::cin >> first >> second;
if(first > second) {
	std::cout << "first more";
}
else {
	if(second > first) {
		std::cout << "second more";
	}
	else {
		std::cout << "equal";
	}
}
```

В представленной программе, если `first > second` ЛОЖЬ, то выполнение переходит в блок `else`. В данный блок вложена ещё одна конструкция `if else`. Если и там выражение `second > first` ЛОЖЬ, то выполнение переходит в блок `else` и на экран выводится информация о равенстве чисел `equal`.

На блок схеме данную конструкцию можно изобразить следующим образом:



Если в `{ блоке кода }` находится только одно выражение, то фигурные скобки можно опустить:

```
int first, second;
std::cin >> first >> second;
if(first > second)
	std::cout << "first more";
else if(second > first)
	std::cout << "second more";
else
	std::cout << "equal";
```

Таким образом, получается конструкция, позволяющая реализовать множественный выбор. Постепенно, отсекая не нужные варианты, выполнение программы найдёт верный. Если ни один из вариантов не подходит, то выполнится код из последнего блока `else` без `if`.

Последний блок обозначает самый противный случай, если ничто другое не подходит:

```
if
else if
else if
else if
else
```

Обычно фигурные скобки не пишутся между `else` и `if`, а в остальных местах их лучше оставить:

```
int first, second;
std::cin >> first >> second;
if(first > second) {
	std::cout << "first more";
}
else if(second > first) {
	std::cout << "second more";
}
else {
	std::cout << "equal";
}
```

4.3 Оператор сравнения

Рассмотрим программу, которая выдаёт медаль отличникам:

```
int grade = 2;
if(grade = 5) {
	std::cout << "get medal" << std::endl;
}


std::grade << "grade: " << grade;
```

На удивление, медаль будет выдана, не смотря на оценку `2`. Более того, на экране появится `grade: 5`.

Дело в том, что оператор = это оператор присваивания. Значение 5 присваивается переменной grade внутри круглых скобок if(…). Результатом присваивания является само присваиваемое значение, т. е. 5. Любое значение отличное от нуля рассматривается как ИСТИНА. Блок кода {} после if() выполнится если внутри if оказывается true: if(true). Двоечник становится отличником и получает медаль следующим образом:

```
if(grade = 5) => if(5) => if(true)
```

По той же причине на экран не выведется empty set:

```
int set = 0;
if(set = 0) {
        std::cout << "empty set" << std::endl;
}
```

Получаются следующие преобразования:

```
if(set = 0) => if(0) => if(false)
```

Сравнение значений следует производить с использованием оператора сравнения == (два равно без пробелов). Результатом == является true (1) если значения равны и false (0), если значения не равны

int grade = 2;
if(grade == 5) {
std::cout << "get medal" << std::endl;
}
std::grade << "grade: " << grade;
	if(grade == 5) => if(true)
	
	// значение в grade не изменилось и равно 2

int set = 0;
if(set == 0) {
std::cout << "empty set" << std::endl;
}
	if(set == 0) => if(true)
	
	// значение в grade не изменилось и равно 2

Если требуется проверить, что два значения не равны между собой, используйте оператор не равно `!=`. В примере ниже `grade != 5 == true`, поэтому на экране появится `no medal`.

int grade = 2;
if(grade != 5) {
std::cout << "no medal" << std::endl;
}
	if(grade != 5){...} => if(true) {...}

Появится ли `ok` на экране в следующем примере:

```
if(true == 7 > 5) {
    std::cout << "ok";
}
```

Операция `>` имеет более высокий приоритет, чем `==`. Сперва происходит вычисление неравенства `7 > 5`, затем результат сравнения проверяется на равенство с `true`:

```
if (true == 7 > 5) => if (true == true) => if(true)
```

Если не уверены, в каком порядке будут происходить вычисления в сложном выражении, воспользуйтесь круглыми скобками:

```
if(true == (7 > 5))
```

В `C++` не следует использовать двойные неравенства, результат может быть неожиданным:

```
if(5 < 2 < 7) => if(0 < 7) => if(true)
```

Итак, если требуется сравнить два значения на равенство следует использовать `==`.

4.4 Булева алгебра

Проблема двойных неравенств разрешима с использованием логических операторов, которые являются операторами алгебры логики или **булевой алгебры**, названной в честь английского математика Джорджа Буля.

Алгебр существует много разновидностей. В самом простом школьном смысле, в алгебре определяется множество чисел и операции над ними. Простейшие из данных операций:

- сложение
- вычитание
- умножение
- деление

В «обычной» алгебре операторы совершают операции над операндами, т. е. числами. В алгебре логики в качестве операндов выступают высказывания.

Высказывание — утверждение или предложение, про которое можно сказать, что оно ИСТИННО или ЛОЖНО.

Основные операции алгебры логики в порядке уменьшения приоритета сведены в таблицу:

Операция	Союз	Оператор	Пример
Инверсия	НЕ	!	!A
Конъюнкция	ИЛИ	||	A || B
Дизъюнкция	И	&&	A && B

Операнды в алгебре логики принимают только два значения ИСТИНА или ЛОЖЬ, поэтому имеется возможность составить таблицу истинности.

Таблица истинности — таблица, в которой перечислены все возможные комбинации значений операндов и, соответствующее им, значение операции. Представим таблицы истинности для каждого логического оператора.

4.4.1 Инверсия

Унарный оператор, изменяет значение единственного операнда на противоположное. В человеческом языке выражается союзом НЕ.

A	!A
0	1
1	0

4.4.2 Дизъюнкция

Результат ИСТИНА, если хотя бы один операнд ИСТИНА. В человеческом языке выражается союзом ИЛИ.

A	B	A || B
0	0	0
0	1	1
1	0	1
1	1	1

4.4.3 Конъюнкция

Результат ИСТИНА, если все операнды ИСТИНА. В человеческом языке выражается союзом И.

A	B	A && B
0	0	0
0	1	1
1	0	1
1	1	1

4.4.4 Тип bool

Из высказываний с использованием указанных операторов можно составлять сложные выражения.

Рассмотрим высказывания:

```
A = закончился хлеб
B = на улице дождь
```

Утверждение «пойдём за хлебом, когда хлеб закончился и на улице нет дождя» может быть записано следующим образом:

```
A && !B
```

Приоритет у инверсии выше, чем у остальных операций, она выполнится в первую очередь:

```
Закончился хлеб И (НЕ на улице дождь) => Закончился хлеб И на улице нет дождя
```

Результат высказывания сохраняется в переменных типа bool. Переменные данного типа могут принимать только два значения true / false (1 / 0).

С использованием типа bool, приведённый пример на языке C++ запишется следующим образом:

```
bool breadIsOver = true;
bool isRain = false;
bool goForBread = breadIsOver && !isRain; // хлеб закончился и нет дождя
if(goForBread == true) {
        std::cout << "go for bread";
}
```

Условие выполняется, если внутри круглых скобок if() оказывается значение true, это позволяет упростить запись условий (== можно не писать):

```
if(goForBread) {} // действие если идем за хлебом
```

Проверку на ложность можно записать следующим образом:

```
if(!goForBread) {} // действие если не идем за хлебом
if(goForBread == false) {} => if(!goForBread) {} => if(!false) {} => if(true) {}
```

Вернёмся к проблеме записи двойного неравенства if (a < b < c) {…}. Она решается с использованием конъюнкции:

```
if(a > b && a < c) {…}
```

4.5 Побитовые операции

Не следует путать логические операторы `&&` и `||` с операторами `&` и `|` поразрядных операций. Рассмотрим более сложный пример:

```
bool markOfDistinction (int sc) {
	if(sc > 50) {
		std::cout << "get mark";
		return true;
	}
	return false;
}

int main()
{
	int hasMedal, score;
	std::cin >> hasMedal >> score;
	if(hasMedal == true || markOfDistinction(score) == true) {
		std::cout << "get admission privileges";
	}
}
```

Данный фрагмент кода позволяет определить, есть ли у абитуриента привилегии при поступлении в вуз. Привилегии даются в случае, если у кандидата имеется золотая медаль или знак отличия.

Программа как всегда начинает и заканчивает своё выполнение в `{ блоке кода }` внутри функции `main()`. Пусть в качестве ввода поступают значения `0` и `75`. Внутри `if()` проверяется первое условие `hasMedal == true`:

```
if(0 == true || …) => if(false || …)
```

Оно ложно, выполнение программы переходит к проверке второго условия:

```
if(… || markOfDistinction(75) == true)
```

Происходит вызов функции `markOfDistinction()`, которой в качестве аргумента через переменную `sc` передаётся `75` (функции подробно рассматриваются в главе 7):

```
if (sc > 50) => if(75 > 50) = > if(true)
```

На экран выводится `getMark`, функция возвращает `true` и программа переносит своё выполнение обратно в функцию `main()`, в то место, откуда был произведён вызов функции `markOfDistinction(score)`:

```
if(… ||  markOfDistinction(75) == true) => if(… || true) => if(true)
```

Дизъюнкция `ИСТИНА`, если хотя бы один из операндов `ИСТИНА`, в данном случае абитуриент получает привилегии.

Допустим теперь на вход поступает `1 и 75`

Внутри `if()` вычисляется первый операнд дизъюнкции:

```
if(hasMedal == true || …) => if(1 == true || …) => if(true || …) =>
if(true)
```

Поскольку первое выражение `ИСТИНА`, вся дизъюнкция целиком является `ИСТИННОЙ` и остальные операнды не проверяются. Т.е. в данном случае функция `markOfDistinction()` не вызывается. Проблема в том, что это функция может делать что-то важное и её вызов обязателен.

Существует несколько способов гарантировать вызов важной функции:

1. Смена операндов местами:

```
if(markOfDistinction(score) || hasMedal) {
        std::cout << "get admission privileges";
}
```

обратите внимание, что `== true` писать не обязательно, если переменные можно трактовать как булевские значения `true/false`.

2. Вызов функции до ветвления с кэшированием (сохранением) возвращаемого значения:

```
bool hasMark = markOfDistinction(score);
if(hasMedal || hasMark) {
        std::cout << "get admission privileges";
}
```

3. Использование вместо логической операции ||, поразрядной (побитовой) дизъюнкции |:

```
std::cin >> hasMedal >> score; // 1 75
if(hasMedal | markOfDistinction(score) ) {
        std::cout << "get admission privileges";
}
```

Побитовые операции совершают логическую операцию с каждым битом числа, например:

`8 | 6 = 14:`

```
1000 = 8
0110 = 6
1110 = 14
```

В нашем случае:

`if(hasMedal | markOfDistinction(75)) => if(true | true) => if(1|1) => if(1) => if(true)`

Поразрядные операции позволяют реализовать "Битовые флаг", с которыми следует ознакомиться самостоятельно.

4.6 Логические выражения

С использованием логических операций можно записывать сложные логические выражения.

Рассмотрим пример:

```
int age= 28;
int cash = 15;
int hourPm = 21;
bool imNotRabbit =false;
bool goByBus = hourPm < 22 && (cash > 35 || !imNotRabbit ||  age <= 6);
```

В представленной программе задаётся возраст `age`, количество денег `cash`, текущее время после полудня `hourPm`, информация о том, что пассажир не является зайцем `imNotRabbit`.

В переменную `goByBus` помещается `true` если ожидающий на остановке потенциальный пассажир сможет уехать на автобусе. Это возможно:

- пока время не достигло `22:00`, когда проходит последний на сегодня автобус.
- Помимо времени:
  - в кармане должно быть достаточно денег,
  - либо пассажир может являться ушастым зверем,
  - либо его возраст не достиг 7 лет.

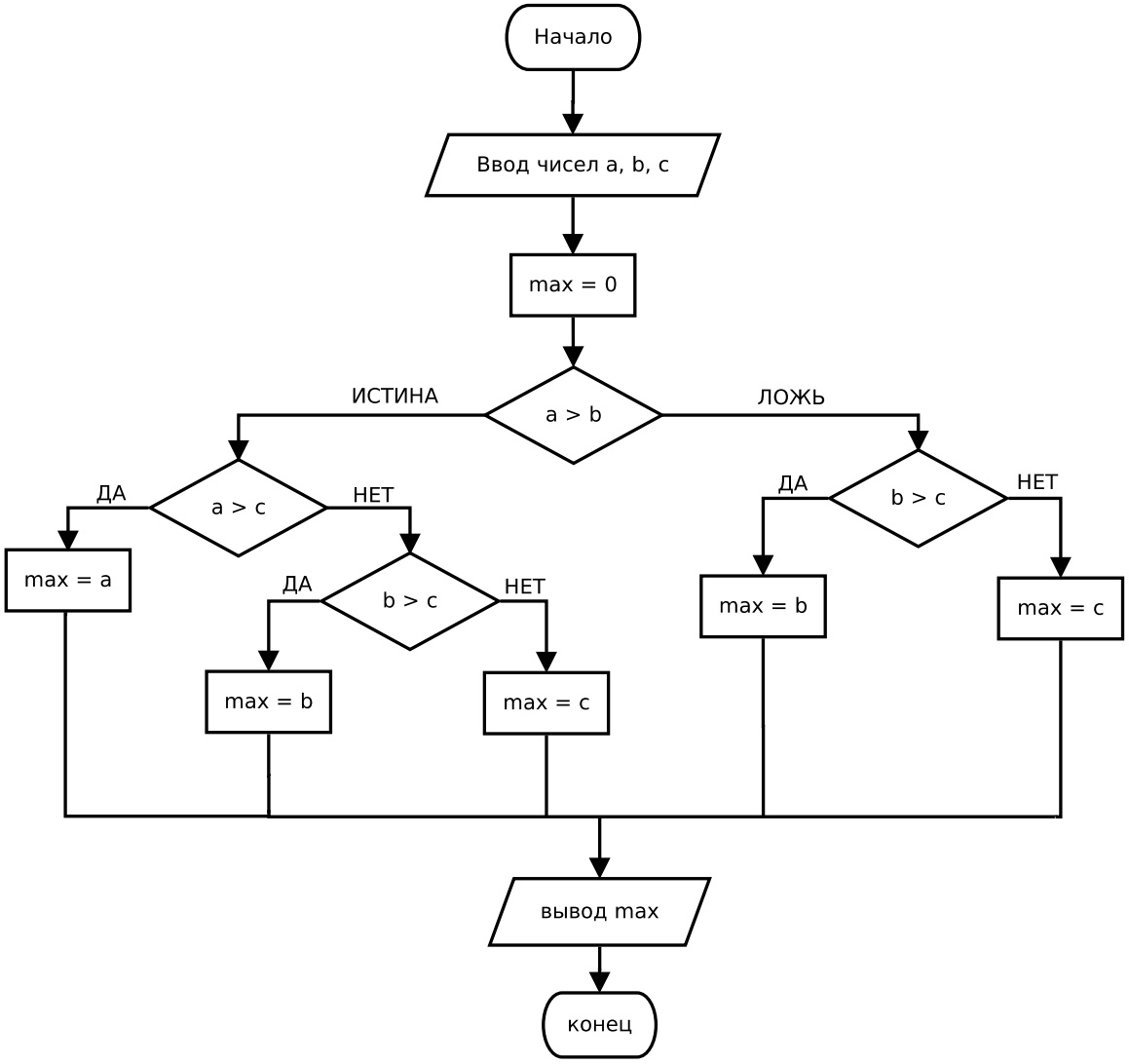
В разобранном примере `goByBus` присваивается значение `true`. Время ещё раннее. Несмотря на то, что пассажир взрослый и не хватает денег, счастливчика спасёт от пешей прогулки, что он является зайцем. Обратите внимание, объявление переменной `imNotRabbit` переводится как `яНеЗаяц = false`. В выражении «`зайцовость`» проверяется командой `!imNotRabbit`, что означает `НЕ яНеЗаяц`.

Итого:

- если хотите сказать `И` пишите `&&`
- если хотите сказать `ИЛИ` пишите `||`
- если хотите сказать `НЕ` пишите `!`
- Помните, что сперва выполняется `!`, затем `&&` и в последнюю очередь `||`.
- Если не уверены в порядке выполнения операций ставьте круглые скобки `()`.

4.7 Вложенные конструкции

Составим блок-схему алгоритма решения следующей задачи: "Пользователь вводит три различных числа. Выведите наибольшее. Операторы && и || использовать запрещено":



```
if(/*...*/) {
        if(/*...*/) {
                //...
        }
        else {
                if(/*...*/) {
                        //...
                }
                else {
                        //...
                }
        }
}
else {
        if(/*...*/) {
                //...
        }
        else {
                //...
        }
}
```

Разумеется с использованием && алгоритм становится проще:

```
if(/*...*/) {
        //...
}
else if (/*...*/) {
        //...
}
else {
        //...
}
```

Примечание: условие if(b>c) в блоке if(a > c) {} else {~~if(b>c) {}~~ } излишне, т. к. если первое условие if(a > b) {…} выполнено, то число b уже не максимально и проверять его не нужно.

Глава 5. Циклы

Линейный алгоритм за сколь-нибудь осмысленное время можно выполнить и без компьютера. Другое дело, когда некоторые действия необходимо повторять множество раз. Если человек с рождения без перерыва на сон и приём пищи будет отсчитывать каждую секунду по одному разу, то когда он досчитает до миллиарда, пройдёт более 32 лет. А представьте, если необходимо не просто считать, а выполнять какие-то действия? Компьютер же с этой задачей справится за считанные секунды, минуты, часы, дни, все зависит от сложности действия, которое необходимо повторять.

Цикл — алгоритмическая конструкция, которая позволяет повторить некоторую последовательность выражений определённое количество раз.

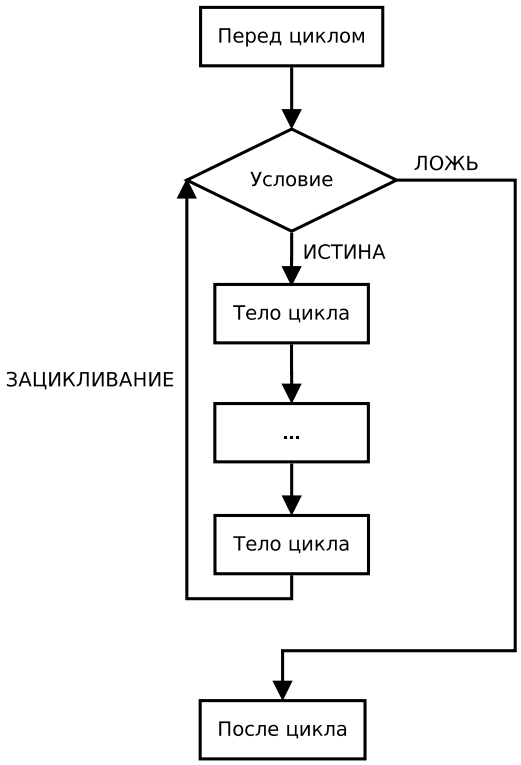
5.1 Цикл while

Простейший цикл называется `while`. Повторяющиеся команды располагаются в `{ блоке кода }`, которой принято называть телом цикла `{ loop body }`. Тело цикла повторяется пока условие `(/*condition*/)`, записанное в круглых скобках, остаётся истинным:

```
while(/*contition*/) {
    // loop
    // ...
    // body
}
```

Итерация — один повтор (проход, выполнение) тела цикла `{ loop body }`.

На блок схемах цикл `while () {}` изображается условием, к которому стрелка возвращается после тела цикла:



Решим задачу "Творческая личность": Художник нарисовал за первый месяц одну картину. Сколько картин художник нарисует за год, если каждый месяц количество нарисованных картин увеличивается в два раза.

Сперва необходимо вникнуть в задачу и произвести её **формализацию**. Под формализацией понимается запись условия задачи в более строгой математической форме:

```
curr_pic = 1 – количество картин, нарисованных в текущем месяце
month = 1 – номер текущего месяца
total_pic = ? - количество картин, нарисованных за год
```

После этого можно приступать к описанию алгоритма:

1. Запишем условие: объявим переменные `curr_pic` и `month` и проинициализируем их.
2. Заведём переменную, хранящую количество картин на текущий момент `total_pic = 0`.
3. Присвоим данной переменной количество картин, нарисованных за первый месяц `total_pic= curr_pic`.

4. Увеличим номер месяца на 1.
5. Пока год не закончился `(month <= 12)`:
   - Умножим количество картин, нарисованных в прошлом месяце на два `curr_pic= curr_pic * 2`, получим в `curr_pic` количество картин, нарисованных в текущем месяце.
   - Прибавим количество картин, нарисованных в текущем месяце, к общему количеству с начала года `total_pic = total_pic + curr_pic`.
   - Увеличим номер месяца `month = month + 1`.
6. Перейдём к п. 5.
7. Выведем `total_pic` на экран.

Алгоритм реализуется с использованием языка программирования, например, `C++`:

```
#include <iostream>
int main() {
	int curr_pic = 1, month = 1, total_pic = 0;
	total_pic = curr_pic;
	++month;
	while(month <= 12) {
		curr_pic *= 2;
		total_pic += curr_pic;
		++month;
	}
	std::cout << total_pic << std::endl;
}
```

Часто при использовании `while()`, возникает ситуация бесконечного цикла. Если ни одна из переменных, указанных в круглых скобках `while()` не изменяется внутри тела цикла `{loop body}`, то `{loop body}` будет повторяться вечно. Либо не повторится ни разу, в том случае если условие `while()` изначально будет ложно `while(false)`.

Иногда цикл намеренно делают бесконечным. Например, "цифровой термометр должен постоянно измерять температуру и выводить её на дисплей, пока не сядет батарейка":

```
while(true) {
	double t = measure_t();
	std::cout << t;
}
```

Можно записать и в одну строчку:

```
while(true) {
	std::cout << measure_t();
}
```

Конструкция `while() {...}` похожа на «зацикленное ветвление» `if() {...}`.

5.2 Операторы break и continue

Рассмотрим программу "Копилка": Человек откладывает на обучение каждый месяц pig_bank рублей только в том случае, если заработная плата в текущем месяце в два раза превысила значение pig_bank. Если в качестве зарплаты вводится отрицательное число, считается, что человек сдался, отправился обогащать банки и взял кредит. Копилка разбивается и на экран выводится накопленная сумма и количество месяцев, в течении которых происходило накопление. Так же копилка разбивается, если человек накопил goal рублей.

Имена для переменных и структуру программы необходимо составлять таким образом, чтобы код был понятен без комментариев и пояснений. Код следует делать **самоописываемым**:

```
int	pig_bank = 0,
	money_box = 0,
	salary = 0,
	goal = 0,
	month = 0;
std::cin >> pig_bank >> goal;
```

```
std::cin >> salary;
while(salary >= 0 || money_box >= goal) {
        ++month;
        if(salary > pig_bank * 2) {
                money_box += pig_bank;
        }

        std::cin >> salary;
}
std::cout << money_box << " " << month;
```

Обратите внимание, что первую зарплату необходимо считать до цикла `while()`, т. к. человек мог отправиться за кредитом, так и не начав копить. Количество месяцев логично увеличивать каждый раз, а не только в месяца с достаточной заработной платой. Для подсчёта количества месяцев был использован уже знакомый паттерн «накопитель». Вспомните, аналогично вычислялась сумма цифр трёхзначного числа в разделе Следование. Данный паттерн используется постоянно.

Перепишем данную программу с использованием бесконечного цикла и двух новых операторов `break` и `continue`.

- Оператор `break` прерывает цикл. Если выполнение программы оказывается в строке `break` цикл немедленно завершается. Выполнение переходит к выражению сразу после цикла.
- Оператор `continue` прерывает текущую итерацию (повтор). Если выполнение программы оказывается в строчке `continue` выполнение программы переходит к проверке условия `while()`. Команды расположенные ниже `continue` на данной итерации пропускаются.

```
int     pig_bank = 0,
        money_box = 0,
        salary = 0,
        goal = 0,
        month = 0;
```

```
std::cin >> pig_bank >> goal;

while(true) {
        std::cin >> salary;
        if(salary < 0) {
                break;
        }
        ++month;
        if(salary < pig_bank * 2) {
                continue;
        }

        money_box += pig_bank;
}

std::cout << money_box << " " << month;
```

В представленном коде, создаётся бесконечный цикл, в котором:

- Первым делом считывается жалование.
- Если оно меньше нуля, выполняется команд `break`, т. е. цикл завершается.
- Если больше или равно нулю, выполнение программы проскакивает `break` и увеличивается количество месяцев.
- Далее проверяется возможность сделать накопления в текущем месяце.
- Если заработной платы не достаточно, то срабатывает оператор `continue`. Выполнение программы на данной итерации не доходит до `money_box += pig_bank`, а переходит к `while(true)`.
- Все повторяется, заново считывается зарплата в следующем месяце.

В случае с вложенными циклами, команда `break` производит выход только из текущего цикла. Внешний цикл продолжает своё выполнение:

```
while(...) {
    ....
    while(...) {
        ....
        if(...) {
            ...
            break;
        }
        ....
    }
}
```

Если требуется осуществить выход из обоих циклов, следует добавить переменную, которая будет передавать внешнему циклу сигнал из внутреннего цикла о выходе:

```
while(...) {
    ...
    bool is_exit = false;
    while(...) {
        ...
        if(...) {
            ...
            is_exit = true;
            break;
        }
        ...
    }
    if(is_exit == true) {
        break;
    }
    ...
}
```

5.3 Разбор числа по цифрам. Поиск минимума

Разберём по цифрам произвольное число любой разрядности. Вспомним, что операция `%10` позволяет узнать последнюю цифру, а операция `/10` отсечь последнюю цифру. В качестве примера выведем все цифры числа в столбик:

```
int num;
std::cin >> n;
while (num > 0) {
    int last_digit = num % 10;
    cout << last_digit << std::endl;
    num /= 10;
}
```

В представленной программе:

- Пользователь вводит число
- Первая строка в теле цикла, копирует последнюю цифру в переменную `last_digit`.
- В конце итерации последняя цифра отбрасывается от числа `num /= 10`.
- Описанные действия происходят пока цифры не закончились, т. е. пока `num > 0`.

Наиболее часто встречаемой задачей является поиск минимального значения. Рассмотрим реализацию паттерна `"поиск минимума"` на примере следующей задачи: `"пользователь вводит число, выведите минимальную цифру числа"`.

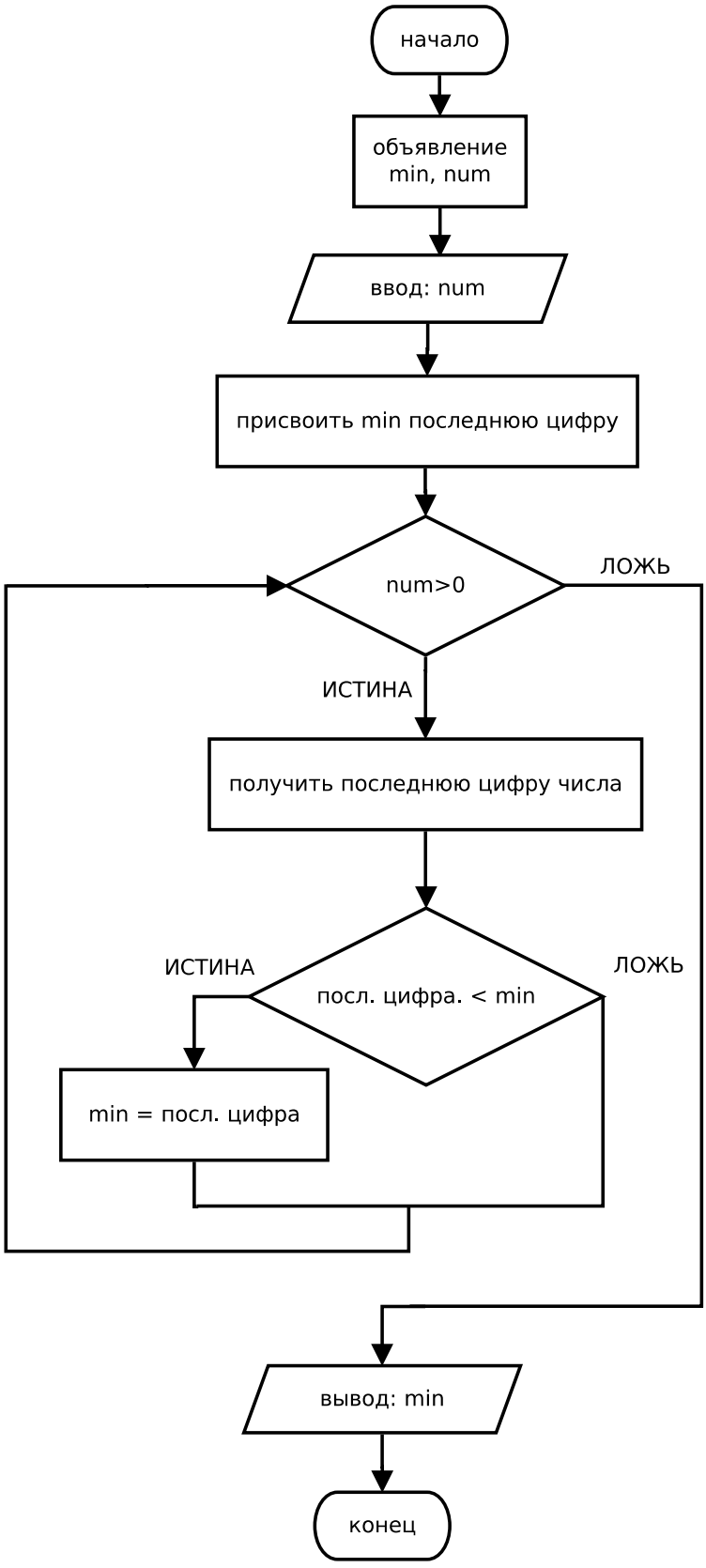
- Логично для хранения минимальной цифры объявить отдельную переменную `min`.
- Изначально предполагаем, что последняя цифра является минимальной.
- В цикле перебираем остальные цифры числа.
- Если попадается цифра меньше значения, хранимого в переменной `min`, заменяем его на текущую цифру.

Блок-схема алгоритма поиска минимальной цифры представлена на следующей странице. Напишите программу, реализующую этот алгоритм самостоятельно.

Можете дополнительно, помимо значения минимальной цифры определить её позицию — номер разряда. Разряды отсчитываются справа налево, начиная с нуля. Если в числе несколько цифр, имеющих минимальное значение, найдите позицию первой такой цифры. Например, для числа 2523, минимальная цифра 2, а позиция минимальной цифры 1.

Подсказка: объявите перед циклом две переменные `curr_pos`, `min_pos`.

Блок-схема алгоритма поиска минимальной цифры числа, введённого пользователем



5.4 Область видимости

Переменная `last_digit` каждую итерацию объявляется внутри цикла. Поскольку область видимости переменной ограничена фигурными скобками `{int a;}`, в которой переменная объявлена, после цикла её уже не существует. Выражение `cout << last_digit`, после цикла `while() {...}` приведёт к ошибке.

И это здорово. После цикла, числа в `num` уже не осталось. Там хранится ноль. Странно, если после цикла имелась бы переменная, хранящая последнюю цифру не существующего числа. Стремитесь переменные объявлять внутри того блока кода `{...}`, в котором они используются. Другими словами, делайте их область видимости наиболее **локальной**. Соблюдение данной рекомендации позволит избежать нелепых логических ошибок, которые, как известно, трудно поддаются обнаружению и исправлению.

Может показаться, что объявлять каждый раз новую переменную внутри затратно с точки зрения производительности. Скорее всего компилятор оптимизирует машинный код таким образом, что переменная будет объявлена до цикла. В худшем случае, если компилятор даст слабину, производительность современных ЭВМ позволяет объявлять целочисленные переменные внутри цикла без серьёзных последствий. Логичность и структурированность кода иногда бывает важнее нескольких тактов машинного времени.

5.5 Цикл for

Цикл `while() {}` следует использовать, если существует некоторая переменная, которая выполняет функцию «критерия» для продолжения цикла. Такую функцию в примере с «копилкой» выполняли переменные `salary` и `total_pig`.

Если количество повторов заранее известно, логичнее использовать цикл `for(;;)`. Цикл `for` достаточно универсальный. В общем случае вместо одного условия, требуется перечислить три выражения (все они не обязательны), каждое из которых отделяется точкой с запятой:

```
for(инициализирующее выражение; условие продолжения; выражение в конце итерации)
{
      // loop...
      // ...body
}
```

Работа цикла происходит по следующему алгоритму:

1. Выполняется инициализирующее выражение.
2. Проверяется условие продолжения, если ЛОЖЬ цикл завершается.
3. Выполняется тело цикла `{ loop body }`.
4. Выполняется выражение в конце итерации.
5. Переход к п. 2.

Обратите внимание на следующие аспекты:

- инициализирующее выражение выполняется обязательно, причём только один раз.
- После инициализации происходит проверка условия продолжения, т.е. тело может ни разу не выполниться.
- Выражение в конце итерации выполняется после тела цикла.

Самый распространённый вариант применения цикла `for` в качестве цикла со счётчиком.

В данном случае алгоритм работы можно описать следующим образом:

```
for(начальное значение счётчика; проверка счётчика; изменение счётчика) {
      // loop...
      // ...body
}
```

1. Счётчику задаётся начальное значение.
2. Происходит проверка счётчика, если `ЛОЖЬ` => счётчик досчитал и цикл завершается.
3. Выполняется тело цикла `{ loop body }`.
4. Счётчик изменяется.
5. Переход к п. 2.

Цикл со счётчиком применяется, когда количество итераций заранее известно из логических соображений или контекста программы, например, значение вводится с клавиатуры.

Вернёмся к задаче `"творческая личность"`: `Художник нарисовал за первый месяц одну картину. Сколько картин художник нарисует за год, если каждый месяц количество нарисованных картин увеличивается в два раза.`

Напомним, как выглядело решение:

```
int curr_pic = 1, month = 2, total_pic = 0;
total_pic = curr_pic;
while(month <= 12) {
        curr_pic *= 2;
        total_pic += curr_pic;
        ++month;
}
std::cout << total_pic << std::endl;
```

Переменная `month` в программе используется с единственной целью отсчитать 12 месяцев.

В алгоритме:

- задаётся номер месяца = 2 (сразу два, т. к. количество картин из первого месяца учитывается перед циклом),
- происходит проверка месяца,
- выполняется тело цикла,
- в конце итерации значение месяца изменяется.

Программа полностью реализует алгоритм работы цикла `for`. Перепишем её с использованием `for`:

Цикл for	**Цикл while**

int curr_pic = 1, total_pic = 0;
total_pic = curr_pic;
for(int month = 2; month <= 12; ++month) {
curr_pic *= 2;
total_pic += curr_pic;
}
std::cout << total_pic << std::endl;
	int curr_pic = 1, month = 2,
	total_pic = 0;
	total_pic = curr_pic;
	while(month <= 12) {
	curr_pic *= 2;
	total_pic += curr_pic;
	++month;
	}
	std::cout << total_pic << std::endl;

Преимущества цикла `for`:

- Код стал более структурированным.
- Код стал более логичным.
- Код занимает на две строчки меньше.
- Область видимости переменной `month` теперь ограничена телом цикла `for() {…}`.
- Никто никогда не забудет дописать `++month` и не получите бесконечного цикла, что часто происходит при использовании цикла `while(){}`.

Всегда, когда требуется что-то подсчитать известное количество раз используйте цикл `for()`. Цикл `for()` применяется в большинстве случаев. Применение цикла `while(){}` обычно связано с умственным усилием и решением какой-то нестандартной задачи, вроде разбора числа по цифрам.

5.6 Основные паттерны цикла for

Наиболее часто встречаемые варианты использования цикла `for()` приведены в таблице. Попробуйте понять ход их работы без дополнительных разъяснений

Паттерн	**Программный код**

Цикл из N повторов, прямой счёт (от 0 до N)	for(int i = 0; i < N; ++i) {
	std::cout << i;
	}

Цикл из N повторов, обратный счёт (от N до 1)	for(int i = N; i > 0; --i) {
	std::cout << i;
	}

Цикл от a до b с шагом s (арифметическая прогрессия)	for(int x = a; x <= b; x += s) {
	std::cout << x;
	}

Геометрическая прогрессия со знаменателем q	for(int x = a; x <= b; x *= q) {
	std::cout << x;
	}

Бесконечный цикл	for(; ;) { }

5.7 Паттерн «Флаг»

Рассмотрим ещё один часто встречающийся шаблон, на примере следующей задачи: "Пользователь вводит N чисел, вывести информацию о том, встретилось ли среди введённых чисел отрицательное".

Будем использовать так называемую **флаговая переменную**. Паттерн заключается в следующем:

- Объявляется флаговая переменная типа `bool`, которая будет сигнализировать о выполнении необходимого условия. Изначально флаговой переменной присваивается значение `false`.
- Выполняются некоторые действия среди котры требуется произвести проверку.
- Если подходящие условие выполнилось, «флаг поднимается» — флаговой переменной присваивается значение `true`.
- После завершения действий проверяем флаг и делаем выводы.

Реализация программы на языке `C++`:

```
int n, N;
bool hasNegative = false;
std::cin >> N;
for(int i = 0; i < N; ++i) {
	std::cin >> n;
	if(n < 0) {
		hasNegative = true;
	}
}

if(hasNegative == true) {
	std::cout << "there were negative numbers";
}
```

Иногда логичнее изначально держать флаг поднятым, т. е. равным `true`, а при выполнении условия сбрасывать его значение в `false`.

Глава 6. Массивы

Следование, ветвление и циклы — конструкции, которые позволяют реализовать любой алгоритм. Алгоритмы обрабатывают некоторые данные, в результате получаются новые данные. Ранее была рассмотрена обработка данных, хранящихся в переменных. Каждая переменная хранит одно значение определённого типа. Типы, которые были рассмотрены, сведены в таблицу:

Тип переменной	**Хранимое значение**
int	Целое число
double	Вещественное (дробное) число
bool	Логическое значение true или false

6.1 Проблема множества данных

Допустим, в классе 5 учеников и у каждого имеется `id` — уникальный номер. Тогда для сохранения оценок учащихся класса требуется создать пять переменных:

```
int grade_1 = 4;
int grade_2 = 5;
int grade_3 = 3;
int grade_4 = 4;
int grade_5 = 3;
```

Вычисление среднего арифметического при этом будет выглядеть следующим образом:

```
double average = (grade_1 + grade_2 + grade_3 + grade_4 + grade_5) / 5.0;
```

Обратите внимание:

- слагаемые числителя заключены в круглые скобки
- деление осуществляется не на целое число `5`, а на дробное число `5.0`
- переменная `average` имеет тип `double`

Код, который подсчитывает количество отличников выглядит следующим образом:

```
int num_five = 0;
if(grade_1 == 5) {
        ++num_five;
}
if(grade_2 == 5) {
        ++num_five;
}
if(grade_3 == 5) {
        ++num_five;
}
if(grade_4 == 5) {
        ++num_five;
}
if(grade_5 == 5) {
        ++num_five;
}
```

Согласитесь, не очень лаконично. Тем более в классах обычно по 30 - 35 человек. Представьте как бы выглядел представленный пример. Или как преобразилась бы программа, определяющая средний балл за ЕГЭ всех учащихся России?

Переменная может хранить, только одно значение, а назначение обрабатывать большие объёмы информации. Хранение множества однотипных значений осуществляется в массивах.

6.2 Определение массива

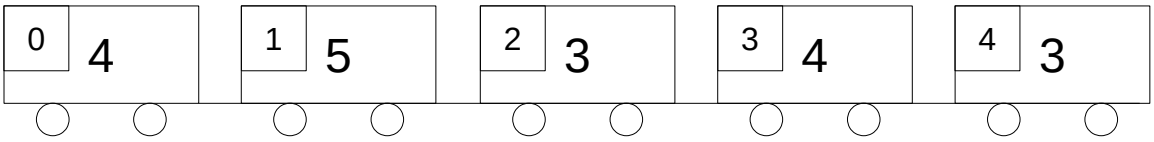
Дадим несколько определений понятию массив:

1. Упрощённое определение

Массив — это состав из вагонов:

- У состава имеется название.
- Каждый вагон хранит одно значение (в нашем случае - целое число).
- Каждый вагон имеет прямоугольную табличку с номером.
- Вагоны нумеруются по порядку, начиная с нуля.

На рисунке показан состав, хранящий оценки учащихся класса из 5-ти человек



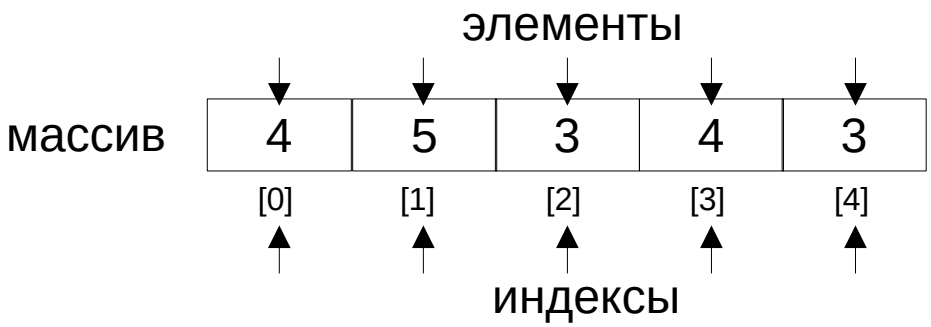
Обратите, внимание, что нумерация вагонов в составе начинается с нуля. Номер первого вагон `НОЛЬ`, а не единица. В составе пять вагонов, но номер последнего `4` — на единицу меньше, чем общее количество вагонов, т. к. отсчёт вагонов ведётся с нуля.

2. Более-менее классическое определение

Массив — переменная, которая хранит в упорядоченном виде много значений одного и того же типа. У массива имеется:

- Имя массива — используется для обращения к массиву.
- Элементы массива — ячейки, которые хранят значения, каждая ячейка хранит одно значение (в нашем случае - целое число).
- Индексы — номера ячеек, используются для доступа к элементам массива
- Индексы элементов идут по порядку, начиная с нуля. Индекс первого элемента массива равен нулю. Индекс последнего элемента на единицу меньше количества элементов в массиве.

Схематично массив изображается следующим образом:



Сравнение упрощённого и классического определения понятия массив

Упрощённое определение	**Более-менее классическое определение**
Состав из вагонов	Переменная, хранящая много значений
Название состава	Имя массива
В каждом вагоне хранится одно значение	В каждом элементе хранится одно значение
Таблички с номером	Индексы
Нумерация с нуля	Нумерация с нуля

Теперь можно хранить информацию об оценках всех учащихся в одном месте — в массиве.

6.3 Объявление, инициализация и обход

Перед использованием массив необходимо объявить. Объявление массива выглядит следующим образом:

```
int grades[5];
```

Сперва указывается тип значений, которые будут хранится в массиве. В случае оценок — целое число `int`. Далее записывается имя массива. Сразу за именем, в квадратных скобках располагается количество элементов в массиве — **длина массива**.

После объявления массива, квадратные скобки используются для доступа к элементам массива. Объявим и заполним массив оценками:

```
int grades[5];
grades[0] = 4;
grades[1] = 5;
grades[2] = 3;
grades[3] = 4;
grades[4] = 3;
```

Внешне работа с массивом похожа на работу с обыкновенными переменными. Просто массив хранит множество элементов, доступ к которым можно получить, указав в квадратных скобках номер — **индекс элемента массива**.

Если значения элементов заранее известны, используют компактную форму записи:

```
int grades[5] = { 4, 5, 3, 4, 3 };
```

Выражение с фигурными скобками `{4, 5, 3, 4, 3}` справа от присваивания называется - **инициализатор**. В инициализаторе через запятую указываются значения, которые помещаются в массив по порядку. Количество значений в инициализаторе должно совпадать с количеством элементов в массиве, указанном в квадратных скобках.

Для обхода всех элементов массива используют цикл `for`. Например, заполнение массива с клавиатуры выглядит следующим образом:

```
int grades[5];
for(int i = 0; i < 5; ++i) {
    std::cin >> grades[i];
}
```

Разберём подробно данный пример:

- Цикл `for` перебирает индексы элементов массива. Именно поэтому переменную-счётчик обычно называют `i` (от англ. `index`).
- Переменная `i` изначально инициализируется `НУЛЁМ`. Это логично, т. к. требуется заполнить весь массив целиком, а индекс первого элемента есть `НОЛЬ`.

- Цикл выполняется пока `i < 5`, т. е. последний раз тело цикла выполнится для значения `i = 4`. Переменная `i` позволяет в теле цикла перебрать все элементы массива.
- В теле цикла введённое пользователем значение сохраняется в `i-ом` элементе массива.

Цикл `for` перебирает индексы элементов массива, которые используются для последовательного доступа ко всем элементам массива внутри тела цикла.

6.4 Основные алгоритмы для работы с массивами

Алгоритм работы с массивом описывается следующим образом:

1. Объявление массива.
2. Инициализация массива.
3. Использование цикла `for` для обхода всех элементов массива.
4. Выполнение с каждым элементом необходимого действия.
5. Вывод результата на экран.

Рассмотрим основные алгоритмы, используемые при работе с массивами.

6.4.1. Сумма всех элементов

Вспоминаем паттерн `"накопитель"`. Применительно к массивам алгоритм выглядит следующим образом.

1. Объявление и инициализация переменной «`накопитель`»:

   ```
   int sum = 0;
   ```

2. Объявление и инициализация массива:

   ```
   const int N = 5;
   int arr[N] = {6, 76, 87, 98, 43};
   ```

3. Перебор индексов элементов массива в цикле `for`:

   ```
   for (int i = 0; i < N; ++i) {
   ```

4. Добавление к накопителю `i-го` элемента:

```
    sum += arr[i];
}
```

5. Вывод результата на экран:

```
std::cout << sum << std::endl;
```

Полностью листинг программы выглядит следующим образом:

```
#include <iostream>
int main() {
    int sum = 0;
    const int N = 5;
    int arr[N] = {6, 76, 87, 98, 43};

    for (int i = 0; i < N; ++i) {
        sum += arr[i];
    }
    std::cout << sum;
}
```

Для удобства была введена переменная `N`, хранящая количество элементов в массиве. Данная переменная объявлена с ключевыми словом `const`. Ключевое слово `const` делает переменную **константной**, т. е. неизменяемой. Массив имеет фиксированную длину, следовательно количество элементов должно быть неизменной величиной, известной в момент компиляции.

Приведём пару примеров с ключевым словом `const`.

Компилирование следующего кода приведёт к ошибке:

```
const int N = 5;
N = 6; // error: assignment of read-only variable ‘N’
```

Ошибка переводится следующим образом: «`Переменная N предназначена только для чтения`»

Следующий код также приведёт к ошибке:

```
const int N; // error: uninitialized ‘const N’ [-fpermissive]
```

Константную переменную нельзя изменить, поэтому её требуется инициализировать, т. е. задать ей значение, при объявлении.

6.4.2 Поиск минимального и максимального элементам

Алгоритм поиска минимального и максимального «чего бы то ни было» совпадает с алгоритмом поиска минимальной и максимальной цифры в числе.

Разберём алгоритм поиска минимального значения элемента массива. Перед циклом следует объявить переменную `min` и записать в неё максимально возможное значение. Максимально возможное значения в случае с поиском цифры в числе равно `9`. Значения элементов, хранимых в массиве, могут быть заранее неизвестны. Например, они берутся из файла, по сети или вводятся с клавиатуры. Максимально возможное значение, которое можно сохранить в элемент массива, неизвестно.

Предположим, что первый элемент с индексом `0` является минимальным:

```
int min = arr[0];
```

Если найдётся элемент со значением меньше `min`, то заменим `min` на найденный элемент:

```
const int N = 5;
int arr[N] = {6, 76, 87, 98, 43};
int min = arr[0];
for (int i = 1; i < N; ++i) {
        if(min < arr[i]) {
                min = arr[i];
        }
}
std::cout << min;
```

Счётчик `i` изначально равен `1`, а не `0`, т. к. бесполезно сравнивать `min = arr[0]` с `arr[0]`.

Пусть представленный выше массив `arr` хранит баллы за экзамен для 5 учеников. На практике часто следует искать не само минимальное значение (балл ученика), а индекс минимального элемента (т. е. номер ученика). Зная номер ученика, легко получить его балл:

```
балл = массив[номер ученика]
```

Алгоритм остаётся тем же, только переменную `min` желательно переименовать в `min_index`. Переменная будет выступать в качестве хранилища индекса минимального на текущий момент элемента. Изначально, предполагаем, что элемент с индексом `0` минимальный:

```
int min_index = 0;
```

Индекс `min_index` следует использовать, для доступа к значению минимального элемента следующим образом:

```
arr[min_index]
```

Реализуйте данный алгоритм самостоятельно.

6.4.3. Сортировка. Метод пузырька

Алгоритмов сортировки существует множество. Рассмотрим самый распространённый, однако не самый эффективный `"метод пузырька"` или `"bubble sort"`.

Суть метода в том, что сравниваются два соседних элемента. Если текущий больше предыдущего, то значения элементов меняются местами. Сортировка происходит в несколько этапов. Номер этапа обозначим буквой `i`.

Отсортируем по возрастанию массив:

```
7    3    5    2    4    1
```

`ЭТАП i = 0:`

Полужирным начертанием будем помечать сравниваемые элементы. Буква `[Y]` в конце строки означает, что элементы следует поменять местами.

шаг 1:	**7**	**3**	5	2	4	1	**[Y]**
шаг 2:	3	**7**	**5**	2	4	1	**[Y]**
шаг 3:	3	5	**7**	**2**	4	1	**[Y]**
шаг 4:	3	5	2	**7**	**4**	1	**[Y]**
шаг 5:	3	5	2	4	**7**	**1**	**[Y]**
	3	5	2	4	1	[7]	

Самое большое значение оказалось на своём месте, в конце массива. Заключим его на схеме квадратные скобки, в том смысле, что данный элемент уже сравнивать не нужно. Всего потребовалось 5 шагов, что на единицу меньше количества элементов в массиве.

Проделанные действия записываются следующим фрагментом кода:

```
for(int j = 0; j < N - 1; ++j) {
        if(a[j] > a[j + 1]) {
                int temp = a[j];
                a[j] = a[j + 1];
                a[j + 1] = temp;
        }
}
```

ЭТАП i = 1:

Пройдёмся по полученному массиву ещё раз, однако последнее значение трогать не будем:

	3	5	2	4	1	[7]	
шаг 1:	**3**	**5**	2	4	1	[7]	**остаются на месте**
шаг 2:	3	**5**	**2**	4	1	[7]	**[Y]**
шаг 3:	3	2	**5**	**4**	1	[7]	**[Y]**
шаг 4:	3	2	4	**5**	**1**	[7]	**[Y]**
	3	2	4	1	[5	7]	

Теперь на своём месте уже два самых больших значения 5 и 7, при этом потребовалось на один шаг меньше, чем в предыдущем случае:

```
for(int j = 0; j < N - 1 - 1; ++j) {
        if(a[j] > a[j + 1]) {
                int temp = a[j];
                a[j] = a[j + 1];
                a[j + 1] = temp;
        }
}
```

ЭТАП i = 2:

Пройдёмся по полученному массиву ещё раз:

	3	2	4	1	[5	7]	
шаг 1:	**3**	**2**	4	1	[5	7]	**[Y]**
шаг 2:	2	**3**	**4**	1	[5	7]	**остаются на месте**
шаг 3:	2	3	**4**	**1**	[5	7]	**[Y]**
	3	2	1	[4	5	7]	

Теперь на своём месте уже три самых крупных значения 4, 5 и 7, при этом потребовалось на один шаг меньше, чем в предыдущем случае.

```
for(int j = 0; j < N - 1 - 2; ++j) {
    if(a[j] > a[j + 1]) {
        int temp = a[j];
        a[j] = a[j + 1];
        a[j + 1] = temp;
    }
}
```

ЭТАП i = 3:

И ещё раз:

	3	2	1	[4	5	7]	
шаг 1:	**3**	**2**	1	[4	5	7]	**[Y]**
шаг 2:	2	**3**	**1**	[4	5	7]	**[Y]**
	2	1	[3	4	5	7]	

```
for(int j = 0; j < N - 1 - 3; ++j) {
    if(a[j] > a[j + 1]) {
        int temp = a[j];
        a[j] = a[j + 1];
        a[j + 1] = temp;
    }
}
```

ЭТАП i = 4:

И ещё раз:

```
                2       1       [3      4       5       7]
шаг 1:          2       1       [3      4       5       7]          [Y]
                1       [2      3       4       5       7]
```

```
for(int j = 0; j < N - 1 - 4; ++j) {
        if(a[j] > a[j + 1]) {
                int temp = a[j];
                a[j] = a[j + 1];
                a[j + 1] = temp;
        }
}
```

Алгоритм завершён, массив отсортирован по возрастанию:

```
[1      2       3       4       5       7]
```

Всего потребовалось 5 этапов `i = 0, 1, 2, 3, 4`. На каждом этапе выполняется один и тот же цикл `for()`, который отличается лишь условием во втором выражении. Заметьте, что количество итераций на каждом этапе определяется выражением `j < N - 1 - i`, где `i` — номер этапа. Таким образом, следующий цикл:

```
for(int j = 0; j < N - 1 - i; ++j) {
        if(a[j] > a[j + 1]) {
                int temp = a[j];
                a[j] = a[j + 1];
                a[j + 1] = temp;
        }
}
```

следует повторить 5 раз для `i = 0, 1, 2, 3, 4`, что на единицу меньше количества элементов в массиве.

Итого получаем готовый алгоритм:

```
for(int i = 0; i < N - 1; ++i) {
        for(int j = 0; j < N - 1 - i; ++j) {
                if(a[j] > a[j + 1]) {
                        int temp = a[j];
                        a[j] = a[j + 1];
                        a[j + 1] = temp;
                }
        }
}
```

Алгоритм называется "метод пузырька", т. к. на каждом этапе вверх поднимается самое большое значение. Примерно также в стакане лимонада сперва поднимаются самые крупные пузырьки.

6.5 Многомерные массивы

Многомерные массивы похожи на одномерные, за тем исключением, что у них имеется более одного индекса. Самые распространённые многомерные массивы содержат два индекса, т. е. два измерения. Для обхода всех элементов необходимо использовать вложенные циклы:

```
const int rows = 2, cols = 3,;
int m[rows][cols] = {    {11, 12, 13},
                         {21, 22, 23} };
for(int i = 0; i < rows; ++i) {
        for(int j = 0; j< cols; ++j) {
                std::cout << m[i][j] << "\t";
        }
        std::cout << std::endl;
}
```

Вывод:

```
11      12      13
21      22      23
```

На двухмерные массивы можно смотреть как на:

- Массив массивов. При объявлении первым индексом определяется количество массивов, а вторым индексом количество элементов в каждом подмассиве.
- Матрицу из `rows` строк и `cols` столбцов, подобно полю для морского боя. Первый индекс определяет номер строки, второй индекс номер столбца. Взгляд на двухмерный массив, как на матрицу является самым распространённым и удобным на практике.
- Непрерывный блок памяти, что является самым корректным. Элементы многомерного массива в `C++` расположены последовательно друг за другом, так же как и в одномерном массиве. Сперва идут элементы первой строки, следом за ней элементы второй строки:

Содержимое	11	12	13	21	22	23
Номер ячейки	0	1	2	3	4	5
Индекс строки	[0]	[0]	[0]	[1]	[1]	[1]
Индекс столбца	[0]	[1]	[2]	[0]	[1]	[2]

Значение `21` хранится в элементе с индексом `m[1][0]`. Чтобы вычислить номер ячейки (фактически адрес ячейки памяти), в которой хранится элемент с указанными индексами, необходимо первый индекс (номер строки `i`) умножить на количество столбцов (элементов в подмассиве) и прибавить к нему второй индекс (номер столбца `j`):

```
address[i][j] = i * cols + j
```

Проверим формулу, для всех элементов, в нашем примере, `cols = 3`:

Элемент	Адрес ячейки	Значение в ячейке
m[0][0]	0 * 3 + 0 = **0**	11
m[0][1]	0 * 3 + 1 = **1**	12
m[0][2]	0 * 3 + 2 = **2**	13
m[1][0]	1 * 3 + 0 = **3**	21
m[1][1]	1 * 3 + 1 = **4**	22
m[1][2]	1 * 3 + 2 = **5**	23

Таким образом в памяти хранятся многомерные массивы.

Глава 7. Функции

Функции позволяют разбить большую программу на малые подпрограммы, что приводит к более удобному сопровождению и изменению программного кода. Алгоритмы становятся более понятными, времени и усилий на разработку тратится меньше. Один раз описав алгоритм в функции, можно использовать его столько раз, сколько необходимо.

7.1 Определение и вызов функции

Рассмотрим простейшую программу, которая определяет максимальное из двух чисел:

```
int main() {
	int a = 5, b = 6;
	int max;
	if (a > b) {
		max = a;
	}
	else {
		max = b;
	}
	std::cout << max;
}
```

Алгоритм поиска максимального из двух значений используется довольно часто. Его можно вынести в отдельную подпрограмму `findMax()`:

```
int findMax(int num_1, int num_2) {
	int res = num_1;
	if(num_2 > res) {
		res = num_2;
	}
	return res;
}
int main() {
	int a = 5, b = 6;
```

```
    int max = findMax(a, b);
    std::cout << max;
}
```

Подпрограмма `int findMax(…)` называется функцией. Функция обладает именем, описывает некоторый алгоритм, выполняемый над входными данными и возвращает результат своей работы. Следует различать определение и вызов функции. **Определение функции** – описание того, что она будет делать. **Вызов** – приводит к выполнению, определённой ранее функции.

Определение записывается в глобальной области перед функцией функцией `main()`. Определение начинается с типа возвращаемого значения, далее следует имя функции. После имени в круглых скобках через запятую перечисляются аргументы. **Аргументы** - это переменные с указанием типов, в которые помещаются входные данные при вызове функции. В фигурных скобках записываются команды алгоритма. Функция завершается, когда выполнение программы доходит до ключевого слова `return`. Справа от `return` записывается результат работы функции, который вернётся в вызывающую функцию.

В нашем примере определение описывается следующим образом:

- Функция называется `max`.
- Возвращает значение типа `int`, результатом работы функции является целое число.
- Принимает два аргумента – два целых числа, первое копируется в переменную `num_1`, второе в переменную `num_2`.
- Выполняет алгоритм выбора максимального значения из двух
- Возвращает найденное значение `res` командой `return` в вызывающую функцию

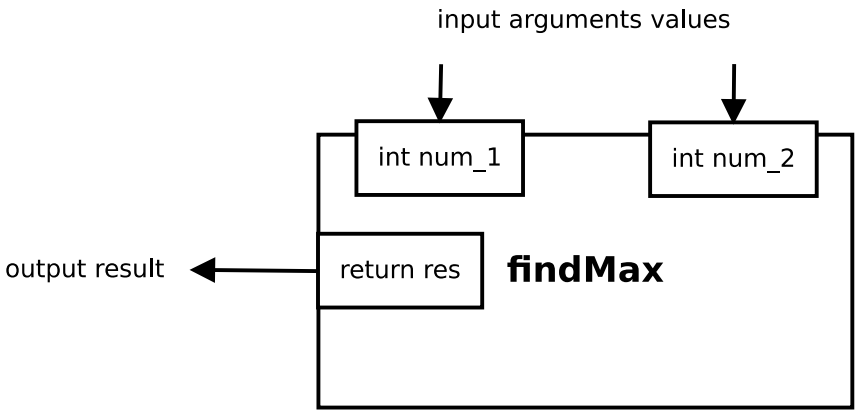
Вызов происходит следующим образом:

- Вызов функции `findMax()` осуществляется во второй строчке функции `main()`. Функция `findMax()` является функцией вызываемой в функции `main()`, а функция `main()`, вызывающей функцию `findMax()`.

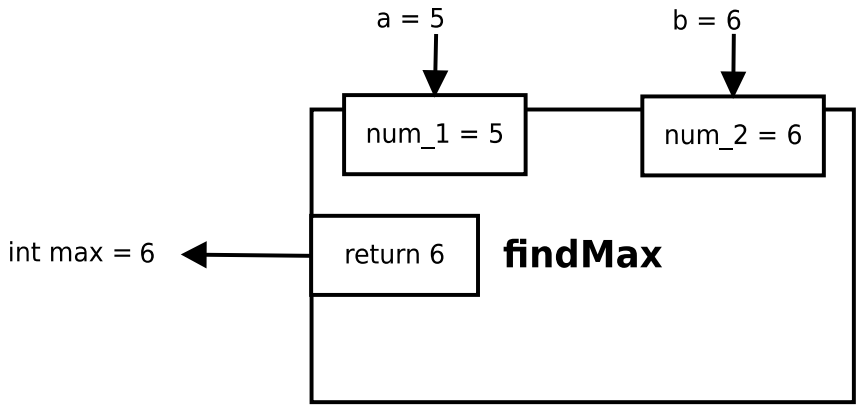
- Для вызова функции записывается её имя, а в круглых скобках через запятую передаются аргументы. Порядок аргументов важен, значение из переменной `a` будет скопировано в переменную `num_1`, а из переменной `b` в `num_2`.
- Выполнение программы переходит в тело функции `findMax()`.
- Исполняется алгоритм, описанный внутри функции `findMax()` с переданными в переменные `num_1` и `num_2` значениями `5` и `6`.
- В результате в переменной `res` оказывается значение `6`.
- Выполнение доходит до строчки `return res`. В этот момент функция завершается, выполнение возвращается в вызывающую функцию `main()`, в то самое место откуда был вызвана `findMax()`.
- Вместо `findMax()` появляется возвращённое значение `6`, которое записывается в переменную `max`:

```
int max = findMax (a, b);
```

Определение функции представим следующей схемой:



Вызов функции будет выглядеть так:



7.2 Объявление функции

Расположим описание функции `findMax()` после функцией `main()`:

```
int main() {
	int a = 5, b = 6;
	int max = findMax(a, b); // error: ‘findMax’ was not declared in this scope
	std::cout << max;
}

int findMax(int num_1, int num_2) {
	int res = num_1;
	if(num_2 > res) {
		res = num_2;
	}
	return res;
}
```

Получим ошибку:

```
error: ‘findMax’ was not declared in this scope
```

Компилятор анализирует текст программы сверху вниз. Сперва обрабатываются команды функции `main()`. В строчке `int max = findMax(a, b)`, компилятор встречает четыре слова:

1. Слово a компилятору известно, оно было объявлено выше, это переменная.
2. Слово b аналогично.

3. Слово `max` объявляется переменной, все хорошо.
4. Ошибка возникает при попытке понять, что такое `findMax()`. Компилятор его до этого не встречал и выдаёт ошибку.

Как быть, если хочется сперва видеть в коде более «важную» функцию `main()`? Следует объявить функцию `findMax()` перед функцией `main()`:

```
int findMax(int num_1, int num_2); // Объявление функции findMax


int main() {
	int a = 5, b = 6;
	int max = findMax(a, b); // error: ‘findMax’ was not declared in this scope
	std::cout << max;
}

// Определение функции findMax()
int findMax(int num_1, int num_2) {
	int res = num_1;
	if(num_2 > res) {
		res = num_2;
	}
	return res;
}
```

Объявление функции состоит только из её заголовка или сигнатуры, который включает:

- тип возвращаемого значения (к сигнатуре не относится),
- имя функции,
- аргументы с указанием их типов в круглых скобках, перечисленные через запятую.

Определение функции дополнительно включает сам код функции, заключенный в фигурные скобки.

Объявление говорит компилятору, что существует функция с конкретным именем, аргументами и возвращаемым значением. Что функция совершает, будет определено позже. Компилятору достаточно данной информации, чтобы проверить выражение

`int max =  findMax(a, b);` на наличие ошибок. Он доверяет объявлению, идёт дальше и находит определение функции `findMax()` ниже.

Объявлений может быть несколько, но определение только одно. Следующий код скомпилируется без ошибок:

```
int findMax(int num_1, int num_2); // Объявление 1
int findMax(int num_1, int num_2); // Объявление 2
int main() {
	int a = 5, b = 6;
	int max = findMax(a, b);
	std::cout << max;
}
int findMax(int num_1, int num_2); // Объявление 3
int findMax(int num_1, int num_2) {
	int res = num_1;
	if(num_2 > res) {
		res = num_2;
	}
	return res;
}
int findMax(int num_1, int num_2); // Объявление 4
```

Объявим функцию, но забудем определить:

```
int findMax(int num_1, int num_2);


int main() {
	int a = 5, b = 6;
	int max = findMax(a, b);
	std::cout << max;
}
```

Запустим команду компиляции:

```
g++ main.cpp
```

Получим следующее сообщение:

```
/usr/bin/ld: /tmp/cc8JZPgm.o: in function `main':
temp.cpp:(.text+0x222): undefined reference to `findMax(int, int)'
collect2: error: ld returned 1 exit status
```

Как ни странно, компиляция прошла успешно. Сообщение на экране относится к ошибке **компоновки**. Компилятору достаточно объявления для совершения своей работы. Компоновщик работает после компилятора и связывает большие откомпилированные части программы воедино. Компоновщик объявлению привязывает конкретное определение. В нашем случае компоновщику не удалось найти определение для функции `findMax()` и он выдал ошибку.

Компиляция, на самом деле, состоит из четырех последовательных этапов:

1. Препроцессинг — подготовка исходного текста.
2. Ассемблирование — преобразование кода, полученного в п.1 в код ассемблера
3. Компиляция кода ассемблера — перевод ассемблерного кода в объектный файл, представляющий собой «кусок» машнинного кода и таблицу символов (объявленные сущности).
4. Компоновка — связь откомпилированных частей (объектных файлов) в единый исполняемый модуль `a.exe`.

Подробно данные этапы рассматриваются в следующих главах. Пока будем считать, что программа `g++.exe` — компилятор, а все четыре этапа — компиляция.

Определение одновременно является и объявлением, поэтому определение `findMax()` перед `main()` без отдельного объявления приводило к успешной компиляции.

Рассмотри пример:

```
int a() {
        b();
        //...
}
int b() {
        a();
        //...
}
```

Как не меняй местами определения `a()` и `b()`, без объявления не обойтись:

```
int a();
int b();
int a() {
        b();
        //...
}
int b() {
        a();
        //...
}
```

Может показаться, что такая конфигурация приведёт к бесконечной рекурсии, но если добавить условия выхода из функций, код вполне работоспособен.

7.3 Применение функций

Любая более менее большая программа состоит из множества функций. Функции используются, чтобы:

1. Вынести в отдельный блок кода часто используемые алгоритмы. Тогда в основном коде программы не придётся писать алгоритм заново, достаточно вызвать функцию. Допустим определена функция извлечения квадратного корня `sqrt()`, выражение нахождения корня квадратного уравнения запишется:

   ```
   double x1 = (- b + sqrt(D)) / (2 * a);
   ```

   Отсутствует необходимость писать программный код для извлечения квадратного корня. В C++ существует библиотека `<cmath>`, содержащая определения готовых математических функций. Необходимо перед или после библиотеки ввода вывода `#include <iostream>` подключить ещё одну библиотеку `#include <cmath>`, тогда для извлечения квадратного корня можно использовать вызов функции `std::sqrt()`.

2. Выполнить определённое действие для объекта. Такие функции называются методами и рассматриваются в следующих главах.

3. Выстроить логическую структуру программы. В большой программе трудно разобраться, но если разбить код на небольшие функции, то понимать и изменять код станет легче. Зачастую такие логические функции ничего не возвращают, тогда в объявлении и определение в качестве возвращаемого значения следует указывать ключевое слово `void`. Функции, которые ничего не возвращают называют **процедурами**. Для таких функций после `return` ничего не указывают, но чаще `return` не пишут вовсе. Функция завершается, когда выполнится её последняя команда. Логическая структура игры, построенная на функциях, выглядит следующим образом:

4.
```
void ShowIntro();
void ShowMenu();
void PlayGame();
void Play();
void Settings();
int main() {
    PlayGame();
}
void ShowIntro() {
        std::cout << "My game Intro animation";
        system("pause");
}
void ShowMenu(){
        std::cout << "Menu: " << std::endl;
        std::cout << "\t1. Play: " << std::endl;
        std::cout << "\t2. Setting: " << std::endl;
        std::cout << "\t3. Exit: " << std::endl;
        int choice;
        std::cin >> choice;
        return choice;
}
void PlayGame() {
    ShowIntro(); // показываем заставку
    int choice = ShowMenu(); // выбираем действие из меню
    if(choice == 1)  {
        Play(); // Играть
    }
    if(choice == 2)  {
        ShowSettings();
    }
    if(choice == 3)  {
        return; // Выход из функции
    }
}
```

В `main()` вызывается единственная функция, запускающая игру. Выполнение программы надолго уходит в `PlayGame()`. Когда данная функция завершит свою работу, выполнение возвращается в `main()` и программа, а вместе с ней и игра, завершается.

7.4 Хорошая функция

Главные правила при разработке функций:

- Функция выполняет одно действие.
- Название функции позволяет понять, какое действие она выполняет.
- Название должно содержать глагол, характеризующий действие.
- Небольшая по размеру. Если функция не умещается на экране, лучше её разбить на несколько простых функций.
- Действие, выполняемое функцией, должно быть как можно более универсальным, т. е. применение функции должно быть возможно в разнообразных случаях.

Имеется задача: `"Пользователь вводит два числа, определить значение максимального числа"`. Следующая функция призвана решить данную задачу:

```
int max() {
	int a;
	int b;
	std::cin >> a;
	std::cin >> b;
	int res = 0;
	if(a > b) {
		res = a;
	}
	else {
		res = b;
	}
	return res;
}
```

```
int main() {
    int max = max();
}
```

Разберём какие из пунктов "хорошей функции" нарушает функция `max()`:

1. Выполняет сразу несколько действий: ввод чисел с клавиатуры и определение максимального значения.
2. Название не содержит глагола, из-за этого в функции `main()` произошёл конфликт имён. Обычно для названия переменных выбирают имя существительное. После слов `int max` переменная считается объявленной, тем самым имя переменной `max` перекрывает внутри `main()` имя функции `max()`. При компиляции появится ошибка:

   ```
   error: ‘max’ cannot be used as a function
   ```

3. Функция достаточно громоздкая, объявление и ввод чисел можно было уместить в два выражения, а в алгоритме избавиться от блока `else`, если для `res` изначально задать значение одной из переменных.
4. Функция не универсальна. Она подходит только для решения конкретной задачи. Функцию нельзя использовать, если числа вводятся не с клавиатуры, а берутся из файла, или если необходимо определить максимальное из трёх чисел. Не универсальность на самом деле исходит из-за нарушения первого пункта:

«одна функция — одно действие»

Хорошая функция выглядит, например, следующим образом:

```
int findMax(int a, int b) {
    return a > b ? a : b;
}
```

Рассмотрим способы её использования:

1. Поиск максимального из двух чисел:

   ```
   int max1 = findMax(5, 6);
   ```

2. Поиск максимального значения из двух переменных:

```
int a = 5, b = 6;
int max2 = findMax(a, b);
```

3. Поиск максимального значения из любого количества переменных:

```
int a = 5, b = 6, c = 3;
int max2 = findMax(a, b);
int max3 = findMax(max2, c);


// Можно записать в одном выражении
int a = 5, b = 6, c = 3, d = 8;
int max3 = findMax(findMax(a, b), c); // максимальное из трёх чисел
int max4 = findMax(findMax(a, b), findMax(c, d)); // максимальное из четырёх чисел
```

4. Поиск максимального элемента массива:

```
const int N = 5;
int a[N] = { 1, 2, 5, 7, 4};
int max = a[0];
for(int i = 1; i < N; ++i) {
        max = findMax(max, a[i]);
}
```

При вызове функции требуется скопировать значения аргументов `max` и `a[i]` в переменные функции `a` и `b`, вернуть результат и выполнение программы обратно в вызывающую функцию. Вызов функции достаточно затратный по времени процесс, поэтому программа будет работать немного медленнее, нежели отказаться от использования функции.

Компилятор имеет встроенный оптимизатор, который делает код более быстрым. Скорее всего оптимизатор компилятора встроит код функции `findMax()` непосредственно в тело цикла `for()` и никакого вызова не произойдёт. Фактически, получится почти тот же машинный код, что и при компиляции программы:

```
const int N = 5;
int a[N] = { 1, 2, 5, 7, 4};
int max = a[0];
for(int i = 1; i < N; ++i) {
        if(a[i] > max) {
                max = a[i];
        }
}
```

Глава 8. Низкий уровень

Было рассмотрено алгоритмическое программирование. Данные сохранялись в массивах и обрабатывались с использованием ветвлений и циклов. Функции были введены для разбиения большой программы на малые независимые части. Грамотное определение и вызов функций позволяет решать поставленные задачи. Всё, что было рассмотрено в том или ином виде присутствует во всех языках программирования. Опустимся немного ниже и посмотрим, как на самом деле выполняется готовая программа. В данном разделе:

- разберёмся с оператором амперсанд и начнём исследовать стек вызовов
- подробно разберём низкоуровневые принципы выполнения программы
- узнаем про указатели, кучу и динамическое выделение памяти

8.1 Оператор амперсанд

Самым многофункциональным оператором языка C++ является амперсанд &. Он может выполнять следующие операции:

1. Поразрядная конъюнкция — если слева и справа от амперсанда добавить операнды:

```
int a = 12, b = 10;
int c = a & b; // 0b1100 & 0b1010 = 0b1000 = 8
```

2. Логическое ИЛИ — если рядом с & написать ещё один &, а слева и справа от от двух амперсандов добавить операнды:

```
int a = 1, b = 0;
bool c = a && b; // 1 && 0 = true && false = false (или 0, то же самое)
```

3. Ссылка — если справа от & написать тип. Ссылки рассматриваются далее:

```
int a = 1;
int& b = a;
```

4. Оператор взятия адреса — если написать операнд слева от переменной:

```
int a = 5;
cout << &a;
```

Рассмотрим подробнее последний случай, в котором амперсанд выступает в качестве унарного оператора — **оператора взятия адреса**.

8.1.1 Исследование оперативной памяти

Запустим программу:

```
#include <iostream>
int main() {
    int a = 5;
    std::cout << &a; // 0x...5f4
}
```

На экране появится следующее значение: `0x...5f4`. Это адрес ячейки оперативной памяти, в которую сохранилось значение `5`. Объявим несколько переменных:

```
int a = 5;
int b = 6;
int c = 7;
std::cout << &a << ": " << a << std::endl;
std::cout << &b << ": " << b << std::endl;
std::cout << &c << ": " << c << std::endl;
```

Получим в консоли следующий вывод:

```
0x...33ac: 5
0x...33a8: 6
0x...33a4: 7
```

Посмотрите внимательно на адреса. Видите закономерность? Продолжение на следующей странице...

Отметим, что следует различать три этапа при программировании:

1. Написание кода в блокноте `hello.cpp`.

2. Компилирование командой `g++ hello.cpp`.

3. Запуск и выполнение готовой программы `a.exe`.

Рассмотрим третий этап. После компиляции получился файл с машинным кодом `a.exe`. В нем имеется несколько команд. Первая команда записывает по адресу `0x...33ac` значение `5`. Вторая команда записывает по адресу `0x...33a8` число `6`. Третья команда записывает по адресу `0x...33a4` число `7`. Никаких типов, никаких переменных, массивов, функций. Ничего этого в готовом `a.exe` нет. Вместо присваиваний простые команды перемещения значений из ячейки в ячейку. Вместо ветвлений и циклов команды перескоков от одной машинной команды к другой.

Машинные команды появились в результате компилирования символов из блокнота, а волшебный оператор амперсанд `&a` позволяет увидеть, где на самом деле «переменные хранятся» в памяти. Разумеется переменных уже нет. Но всё же будем, говорить, что переменные хранятся в конкретных ячейках оперативной памяти, иначе ничего не понять.

Вернёмся к примеру. Адреса ячеек, в которых хранятся переменные, отличаются друг от друга на четыре. Почему? Выведем на экран информацию о том, сколько байт отводится для хранения целого числа типа `int`:

```
std::cout << sizeof(int); // 4
```

Вот и ответ. Четыре байта отводится для хранения значения `a = 5`:

```
0x...33ac
0x...33ab
0x...33aa
0x...33a9
```

Четыре байта отводится для хранения значения `b = 6`:

```
0x...33a8
0x...33a7
0x...33a6
0x...33a5
```

Четыре байта для хранения значения `c = 7`:

```
0x...33a4
0x...33a3
0x...33a2
0x...33a1
```

Следующий код:

```
std::cout << "&a + 1: " << (&a + 1) << std::endl;
```

выдаст значение `0x...33a8`. Стоп. Адрес по которому хранится `a` равен `0x...33ac`. Прибавим единицу `&a + 1 = 0x...33ac + 1`, должно получиться `0x...33ad`, но никак не `0x...33a8`. Результат даже меньше исходного значения. Дело в том, что `&a` не обычное число. Это адрес, по которому начинает храниться значение типа `int`. Прибавление `1` в данном случае, означает переход к ячейке, расположенной сразу после ячеек, отводимых под хранение значения переменной `a`, т. е. `0x...33ac - sizeof(int)`.

Выражение `&a` извлекает адрес первой ячейки оперативной памяти (первого байта), с которого начинает хранится переменная `a`. Чтобы вычислить адрес ячейки в которую будет сохранено значение следующей переменной, из `&a` необходимо вычесть размер типа переменной `a`, т. е. `sizeof(int)`, т. е. четыре.

В общем виде:

```
type a;
(&a + 1) <=> адрес(a) - sizeof(type);
```

В принципе компилятор мог бы раскидать ячейки для хранения значений переменных по всей памяти. Данные и машинные команды перепутались, воцарился хаос, память закончилась, а вместе с ней и век развития информационных технологий. К счастью или нет,

компилятор генерирует машинные команды, которые укладывают значения переменных строго по порядку, как стопку книг на полке. Полка называется **стеком вызовов**. Книги — значения переменных. В больших программах чтение верхних книг происходит наиболее часто. После использования верхние книги убираются, на их место кладутся новые. Таким образом, память не переполняется. Книги которые лежат в самом низу исчезнут с полки, только после закрытия программы. Они самые живучие.

8.2 Выполнение программы

Рассмотрим пример высокоуровневого программного кода. Приведем результат, который может получиться после компиляции программы. Посмотрим на схему процесса в оперативной памяти программы и разберем этапы выполнения машинного кода микропроцессором. Познакомимся с некоторыми инструкциями языка ассемблера и подробно исследуем, что происходит в оперативной памяти в процессе выполнения программы.

8.2.1 Высокоуровневый код

Рассмотрим в качестве примера следующую программу:

```
int f(int c, int d) {
        int e;
        int k;
        e = (c + d);
        return e;
}
int main() {
        int a;
        int b;
        int res;
        a = 2;
        b = 5;
        res= f(a, b);
        return;
}
```

Функция `f` принимает в качестве аргументов два целых числа и возвращает их сумму. Внутри функции `main()` объявляется три переменные, двум из них присваиваются значения, которые передаются в качестве аргументов функции `f()`, результат работы функции заносится в переменную `res`. Переменная `res` в итоге хранит число 7.

Функцию `main()` будем называть **вызывающей**, функцию `f()` **вызываемой**. Вызывающая функция `main()` вызывает вызываемую функцию `f()`.

<u>Предупреждение:</u> Дальнейшее описание, к сожалению, может содержать много неточностей. В реальных системах дела скорее всего обстоят несколько по другому. Но все нижеизложенно не противоречит логике и здравому смыслу, значит имеет место быть. Такая упрощенная модель на первых порах может быть полезна тем, кто стремится разобраться, как высокоуровневый код выполняется компьютером.

8.2.2 Ассемблерный код

Код ассемлера после компиляции программы может выглядеть следующим образом:

```
main:
        LDI     ax, 2
        OUT     [bp – 2], ax
        LDI     bx, 5
        OUT     [bp – 4], bx
        ADI     SP, 2
        PUSH    bx
        PUSH    ax
        CALL    f
        POP     ax
        OUT     [bp – 6], ax
f:
        PUSH    bp
        MOV     bp, sp
        SUB     sp, 4
        IN      ax, [bp + 4]
        IN      bx, [bp + 6]
        ADD     ax, bx
        OUT     [bp – 2], ax
        IN      ax, [bp – 2]
        OUT     [bp + 8], ax
        MOV     sp, bp
        POP     bp
        RET     4
```

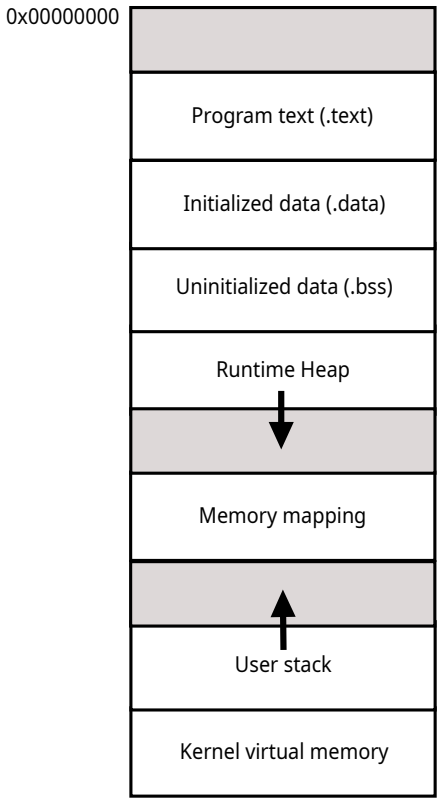
Машинный код (нули и единицы) из ассемблерного получить легко, нужно каждой инструкции, записанной через мнемонику, сопоставить соответствующий двоичный код, при

этом соответствие однозначное: мнемоника — двоичный код. Пока что низкоуровневый код не понятен, главное усвоить, что каждая строка в этом коде представляет собой отдельную инструкцию для микропроцессора, записанную в виде символов — **мнемоник**. Будем считать, что каждая инструкция (строка кода) занимает ровно 2 байта (16 бит), например, `1101101011010011`.

8.2.3 Память программы

Все адреса которые извлекаются оператором `&a`, не являются реальными адресами оперативной памяти. Между программой и физической оперативной памятью располагается виртуальная память.

Операционная система для каждого запущенной программы (процесса) выделяет виртуальное адресное пространство. Программы выполняются в своих «песочницах» и не мешают друг другу. Виртуальную оперативную память для любой программы можно представить следующей схемой:



Ячейка с адресом ноль на схеме представлена вверху. Виртуальная память разделена на следующие блоки:

- В начале виртуальной памяти имеется небольшой блок пустоты, в нем ничего никогда не хранится.
- `Program text` — в данную область копируются машинные команды скомпилированной программы.
- `Initilized data (.data)` — область, в которой хранятся инициализированные глобальные переменные и инициализированные локальные статические переменные.
- `Uninitilized data (.bss)` — область для хранения неинициализированных глобальных переменных и неинициализированных локальных статических переменных.
- `Runtime heap` — динамическая память или куча, область для хранения данных, размещаемых оператором `new`.
- `Memory mapping` — ядро использует этот сегмент для мэппирования (отображения в память) содержимого файлов, применяется при загрузке динамических библиотек типа `.dll`.
- `User stack` — стек вызовов.
- `Kernel virtual memory` — ядро, в ней размещается информация операционный системы, необходимая для работы программы. Например, здесь располагаются специальные таблицы, связывающие адреса виртуальной памяти с реальными физическими адресами оперативной памяти.

Для упрощения изложения и понимания материала, будем считать, что виртуальная память является реальной физической оперативной памятью.

8.2.4 Исполнение программы микропроцессором

Компиляция программы, написанной на языке высокого уровня, выдает машинный код — набор чисел, которые для удобства чтения можно записать в виде символических мнемоник (п. 8.2.2). Машинный код сохраняется на жёстком диске в двоичном виде. После запуска

программы операционная система копирует этот код в оперативную память (т. к. взаимодействие с ней осуществляется быстрее), в сегмент `Program Text (.text)`.

Фрагмент Оперативной памяти с процессом			
Сегмент	Адрес	Содержимое	Пояснение
Program text	0xA0	LDI ax, 2	Инструкции функции main()
	0xA2	OUT [bp – 2], ax	
		...	
	0xAE	CALL f	
		...	
	0xB4	PUSH bp	Инструкции функции f()
	0xB6	MOV bp, sp	
		...	
	0xC8	POP bp	
	0xCA	RET 4	
		...	
		...	

Ячейка оперативной памяти имеет размер 1 байт и располагается по определенному адресу, например, `0xA1` (в таблице ячейка не показана). Каждая инструкция программы занимает в памяти 2 байта, поэтому адреса в таблице увеличивается с шагом `+2`: инструкция `LDI ax, 2` занимает две ячейки оперативной памяти `0xA0` и `0xA1`.

В микропроцессоре `CPU` (`central processor unit`) имеется два важных регистра (ячейки):

1. **Программный счётчик PC (Program Counter)** хранит адрес инструкции, которая будет выполнена следующей.
2. **Регистр команд IR (Instruction Registr)** хранит машинный код текущей инструкции.

Содержимое данных регистров после запуска программы отобразим в таблице:

CPU	
PC	0xA0
IR	0000000000000000

Допустим, команда `0000000000000000` обозначает **микропрограмму микропроцессора — `FETCH`.** Микропрограмма `FETCH` извлекает инструкцию, располагаемую по адресу, записанному в программном счетчике (`PC = 0xA0`), из области `Program text` оперативной памяти в регистр микропроцессора `IR`.

Микропроцессор выполняет программу следующим образом:

1. FETCH копирует инструкцию, расположенную в сегменте памяти Program text, по адресу, указанному в программном счётчике PC, в регистр команд IR.
2. Программный счётчик PC увеличивается на единицу. В нём оказывается адрес следующей инструкции.
3. Микропроцессор выполняет очередную инструкцию, которая расположена в регистре команд IR.
4. Переход к п. 1.

В момент, когда CPU выполняет инструкцию из IR, в PC хранится адрес следующей инструкции. После выполнения текущей инструкции, в IR заносится инструкция по адресу из PC. При этом PC увеличивается и указывает на адрес следующей инструкции.

Некоторые инструкции могут изменять значения программного счетчика PC, переводя выполнение программы (перепрыгивание) к той или иной инструкции (строчке кода). Таким образом, на низком уровне реализуется ветвление, циклы и вызовы функций.

8.2.5 Стек вызовов

Часть памяти процесса отведено под стек вызовов User stack.

Фрагмент Оперативной памяти с процессом			
Сегмент	Адрес	Содержимое	Пояснение
Program text	0xA0	LDI ax, 2	Инструкции функции main()
	0xA2	OUT [bp – 2], ax	
		...	
	0xAE	CALL f	
		...	
	0xB4	PUSH bp	Инструкции функции f()
	0xB6	MOV bp, sp	
		...	
	0xC8	POP bp	
	0xCA	RET 4	
		...	
		...	
User Stack		...	
	0xF7		Стековый фрейм функции main()
	0xF9		
	0xFB		
	0xFD		
	0xFF		

Стек вызовов — место, куда сохраняется информация для возврата из вызываемой подпрограммы (функции) в вызывающую программу (функцию), а также данных (локальных переменных), появляющихся и исчезающих в программе. В нашем примере, вызываемая функция это `f()` с переменными `c, d, e, k`, а вызывающая функция `main()` (та, которая вызывает функцию `f()`) с переменными `a, b, res`.

Размер минимального элемента, который может быть помещен в стек назовем **машинным словом**. Пусть в нашем примере, машинное слово занимает ровно 2 байта.

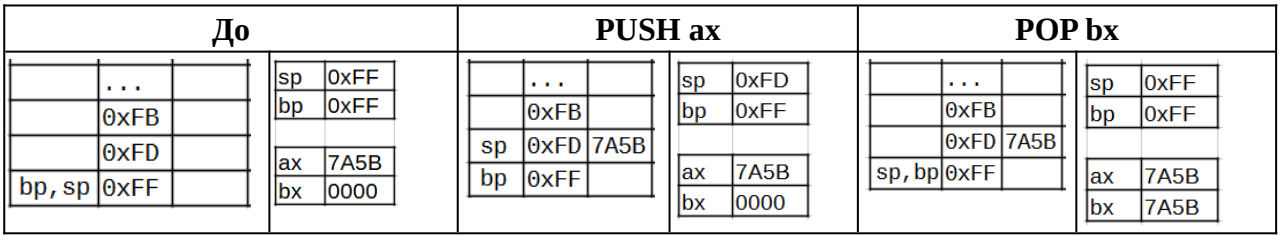
Стек по своей структуре напоминает стопку тарелок. Каждая тарелка — это двухбайтовое число. Стек «растет» снизу вверх: по мере наполнения стека, адреса ячеек оперативной памяти, в которые помещаются значения, уменьшаются.

Работа со стеком реализуется через два специальных регистра. Регистры определяют область оперативной памяти, занятую стеком в настоящий момент: это вершина стека `SP` и дно стека `BP`:

1. **Вершина стека SP (Stack Pointer)** - содержит адрес последнего помещённого в стек машинного слова
2. **База стека BP (Base Pointer)** - содержит адрес дна стека.

Рассмотрим выполнение микропроцессором команды `PUSH ax`, которая заносит значение из регистра данных `ax` в стек. Пусть изначально в `SP` и `BP` записано значение `0xFF`, вершина совпадает с дном, т. е. стек пуст. Команда `PUSH ax`:

- Сперва уменьшает значение указателя `SP` на 2 (длину машинного слова 2 байта): `SP = 0xFF – 2 = 0xFD`.
- Затем копирует данные из регистра `ax` в ячейку, на которую указывает `SP`, т. е. в ячейку с адресом `0xFD`.



Рассмотрим выполнение микропроцессором команды `POP bx`, которая извлекает значение из стека и заносит его в регистр данных `bx`. Команда `POP bx`:

- Сперва копирует данные из стека, расположенные в ячейке, на которую указывает `SP` (т. е. `0XFD`).
- Затем увеличивает значение указателя `SP = 0xFD + 2 = 0xFF` на длину машинного слова (т. е. на 2).

Таким образом, добавление значения в стек приводит к уменьшению `SP` на длину машинного слова (2 байта) (смещение вверх), при изъятии элемента из стека `SP` увеличивается на 2 (смещается вниз). Значение bp изменяется при вызове функции и возврате из функции.

Вся память стека ниже ячейки, на которую указывает `SP`, в т.ч. и ячейка, на которую указывает `SP`, является заполненной частью стека (после команды `PUSH` ячейка с адресом `0xFD` считается заполненной). Все, что выше ячейки, на которую указывает `SP` – свободная область стека (после команды `POP` ячейка с адресом `0xFD` считается свободной). Ячейка на которую указывает `SP` является самой верхней заполненной ячейкой стека.

8.2.6 Описание инструкций ассемблера

Разберем, что делает каждая инструкция ассемблерного кода из п. 8.2.2.

`LDI ax, num` – записывает число num в регистр данных `ax`

`MOV ax, bx` – копирует значение из регистра `bx` в регистр `ax`

`IN ax, addr` – считвает значение из ячейки оперативной памяти с адресом `addr` в регистр `ax`

`OUT addr, ax` – записывает в ячейку оперативной памяти с адресом `addr` значение из регистра `ax`

`PUSH ax` – уменьшает адрес в указателе стека SP на 2 и заносит значение из регистра `ax` в стек

`POP ax` – извлекает значение (разумеется, самое верхнее) из стека в регистр `ax` и увеличивает адрес в указателе стека `SP` на 2.

`CALL addr` – осуществляет вызов подпрограммы (функции), располагающейся по адресу `addr`. Производит два действия:

- Заносит значение из программного счетчика `PC` в стек, чтобы, после завершения вызываемой подпрограммы, выполнение программы вернулось к инструкции, следующей за `CALL`. Напомним, что `PC` хранит адрес памяти, в которой располагается код следующей инструкции программы.
- Копирует адрес первой инструкции вызываемой подпрограммы `addr` в программный счетчик `PC`, переводя выполнение программы микропроцессором в вызываемую подпрограмму.

8.2.7 Подробный разбор выполнения машинного кода и работы стека вызовов.

Взглянем еще раз на полный код программы, занесенной в ячейки оперативной памяти и изначальное состояние стека. Допустим, для общности изложения, что нашу функцию `main()` вызвала некоторая функция `os()` операционной системы. После завершения функции `main()`, следует вернуться обратно в вызывающую функцию `os()`. С этой целью в стек помещены адрес дна стека `BP` для функции `os()` и адрес возврата в функцию `os()` - адрес инструкции из функции `os()`, которая должна быть выполнена после завершения функции `main()`.

main:	
0xA0	LDI ax, 2
0xA2	OUT [bp - 2], ax
0xA4	LDI bx, 5
0xA6	OUT [bp - 4], bx
0xA8	ADI SP, 2
0xAA	PUSH bx
0xAC	PUSH ax
0xAE	CALL f
0xB0	POP ax
0xB2	OUT [bp - 6], ax
f:	
0xB4	PUSH bp
0xB6	MOV bp, sp
0xB8	SUB sp, 4
0xBA	IN ax, [bp + 4]
0xBC	IN bx, [bp + 6]
0xBE	ADD ax, bx
0xC0	OUT [bp – 2], ax
0xC2	IN ax, [bp - 2]
0xC4	OUT [bp + 8], ax
0xC6	MOV sp, bp
0xC8	POP bp
0xCA	RET 4

	Addr	Value	Comments	sp	0xF7
	...			bp	0xFD
	0xE7				
	0xE9				
	0xEB				
	0xED				
	0xEF				
	0xF2				
	0xF4				
	0xF5				
sp	0xF7		фрейм функции main() (локальные переменные)		
	0xF9				
	0xFB				
bp	0xFD	адрес BP в os()			
	0xFF	адрес возврата в os()			

Область стека (между `SP` и `BP`), предназначенная для хранения локальных переменных функции, называется **стековым фреймом функции**. Регистр `SP` указывает на ячейку `0xF7`, таким образом, для хранения локальных переменных `a`, `b`, `res`, выделено три машинных слова (по одному слову на каждую переменную).

Поэтапно разберем выполнение каждой инструкции программы и какие изменения в стеке при этом происходят.

1. Объявление и инициализация переменных

Далее, в левом столбце будут указаны рассматриваемые инструкции, а в правом состояние стека **после** выполнения этих инструкций.

Высокоуровневый смысл понятия **переменная** — область оперативной памяти, обозначенная в программе именем, предназначенная для хранения данных определенного типа. Однако, в ассемблере нет переменных и нет имен. Для размещения данных в стеке, в качестве точки отсчета используется значение регистра `BP`. Значение этого регистра не изменяется, пока выполнение программы находится внутри текущей функции, что удобно. Компилятор в ходе анализа высокоуровневого кода, просто сопоставляет с каждым именем смещение относительно дна стекового фрейма `BP` (адреса, хранящегося в регистре `BP`).

`main:`		
`0xA0`	`LDI ax, 2`	a = 2
`0xA2`	`OUT [bp - 2], ax`	
`0xA4`	`LDI bx, 5`	b = 5
`0xA6`	`OUT [bp - 4], bx`	

	Addr	Value	Comments	sp	**0xF7**
	...			bp	**0xFD**
	0xE7			PC	0xA8
	0xE9				
	0xEB				
	0xED				
	0xEF				
	0xF2				
	0xF4				
	0xF5				
sp	**0xF7**		фрейм функции `main()` (локальные переменные)		
	0xF9	**5**			
	0xFB	**2**			
bp	**0xFD**	адрес BP в os()			
	0xFF	адрес возврата в os()			

Например, для переменной `a` смещение равно `2` (т. е. одно машинное слово), это означает, что значение переменной запишется в самый низ стекового фрейма. Для переменной `b` смещение равно `4`, т. е. её значение запишется следом за `a`. Как вы догадались для переменной `res` зарезервирован адрес `[bp - 6]`. Адресация получается относительная (относительно `BP`). Абсолютная адресация невозможна: никто не в силах гарантировать конкретные ячейки оперативной памяти, которые будут выделены под стековой фрейм вызываемой функции. Стековый фрейм одной и той же функции, вызываемой из разных мест кода, будет располагаться в разных областях оперативной памяти. Соответственно, абсолютные адреса ячеек, в которые будут помещены локальные переменные функции, от вызова к вызову отличаются.

Вернемся к коду ассемблера. Инструкция `LDI` необходима, т. к. в рассмотренном нами наборе команд не существует той, которая позволяла бы записать число в память непосредственно. Имеется только команда `OUT`, которая может скопировать значение из регистра в память . Поэтому предварительно число записывается в регистр `ax`, а уже из него в ячейку оперативной памяти с адресом `[bp - 2]`. Аналогичная ситуация встретиться с командой чтения из памяти `IN`.

2. Выделение ячейки для адреса возврата

Далее идут подготовительные операции перед вызовом функции `f()`. Поскольку функция возвращает значение, в стеке необходимо выделить ячейку, куда это значение будет записано, до того как функция `f()` вернет выполнение программы обратно в `main()`. Это значение после завершения выполнения функции `f()` будет извлечено из стека в переменную `res`.

0xA8	ADI SP, 2

	Addr	Value	Comments	sp	0xF5
	...			bp	**0xFD**
	0xE7			PC	0xAA
	0xE9				
	0xEB				
	0xED				
	0xEF				
	0xF2				
	0xF4				
sp	0xF5	резерв для return_value			
	0xF7		фрейм функции main() (локальные переменные)		
	0xF9	5			
	0xFB	2			
bp	**0xFD**	адрес BP в os()			
	0xFF	адрес возврата в os()			

3. Передача аргументов в функцию f()

Аргументы заносятся в стек в обратном порядке. Сперва второй аргумент (переменная `d`), затем первый (переменная `c`). При выполнении функции `f()` порядок аргументов будет соответствовать прямому, что вы увидите далее.

0xAA	PUSH bx	d = 5
0xAC	PUSH ax	c = 2

	Addr	Value	Comments	sp	0xF2
	...			bp	**0xFD**
	0xE7			PC	**0xAE**
	0xE9				
	0xEB				
	0xED				
	0xEF				
sp	**0xF2**	**2**	c = 2		
	0xF4	**5**	d = 5		
	0xF5	резерв для return_value			
	0xF7		фрейм функции main() (локальные переменные)		
	0xF9	5			
	0xFB	2			
bp	**0xFD**	адрес BP в os()			
	0xFF	адрес возврата в os()			

4. Вызов функции f()

Значение из `PC = 0xB0` (программный счетчик указывает на адрес следующей за `0xAE` инструкции) записывается в стек в качестве адреса возврата, а в программный счетчик заносится адрес первой инструкции функции `f()`. В ассемблерном коде активно используются **метки** — символьное обозначение для адреса инструкции. Фактически в машинном коде хранится смещение относительно адреса текущей инструкции. Оно то и прибавляется к программному счетчику, чтобы получить адрес начала функции `f()`.

Значение, которое инструкция `CALL` занесет в программный счетчик выражается следующим образом: `PC = 0xB0 - 2 + 6 = 0xB4`, где

- `0xB0` – текущее значение программного счетчика, указывает на адрес следующей инструкции
- `-2` – вычитание двойки дает адрес выполняемой в данной момент инструкции `0xB0 – 2 = 0xAE`
- `+6` – прибавление шести смещает адрес на три инструкции ниже (каждая инструкция занимает машинное слово – 2 байта, т. е. две ячейки): `0xAE + 6 = 0xB4`

0xAE	CALL f (CALL 6)	PC – 2 + 6

	Addr	Value	Comments	sp	**0xEF**
	...			bp	**0xFD**
	0xE7			**PC**	**0xB4**
	0xE9				
	0xEB				
	0xED				
sp	**0xEF**	**0xB0 адрес возврата**			
	0xF2	2	c = 2		
	0xF4	5	d = 5		
	0xF5	резерв для return_value			
	0xF7		фрейм функции main() (локальные переменные)		
	0xF9	5			
	0xFB	2			
bp	**0xFD**	адрес BP в os()			
	0xFF	адрес возврата в os()			

5. Пролог функции f()

В прологе формируется стековый фрейм. Сперва сохраняется в стеке дно стекового фрейма вызывающей функции `main()`: команда `PUSH bp` заносит значение `0xFD` из `BP` в стек. После этого значение `SP = 0xED`. Сохранить дно стекового фрейма функции main() необходимо, чтобы в момент возврата в `main()` имелась возможность его восстановления.

Команда `MOV bp, sp` поднимает дно `BP` до уровня вершины стека, теперь вершина стека совпадает с дном.

0xB4	PUSH bp
0xB6	MOV bp, sp
0xB8	SUB sp, 4

	Addr	Value	Comments	sp	0xE9
	...			bp	0xED
	0xE7			PC	0xBA
sp	0xE9		фрейм функции f()		
	0xEB				
bp	0xED	0xFD адрес BP в main()			
	0xEF	0xB0 адрес возврата			
	0xF2	2	c = 2		
	0xF4	5	d = 5		
	0xF5	резерв для return_value			
	0xF7		фрейм функции main() (локальные переменные)		
	0xF9	5			
	0xFB	2			
	0xFD	адрес BP в os()			
	0xFF	адрес возврата в os()			

Инструкия `SUB sp, 4` вычитает из стека число четыре, т. е. `SP = 0xED - 4 = 0xE9`. Формируется стековый фрейм для функции `f()`. Ячейка `[bp - 2]` резервируется для переменной `e`, `[bp - 4]` для переменной `k`. Переменная `k` не несет никакой смысловой нагрузки. Она введена для наглядности, иначе стековый фрейм состоял бы из одной ячейки.

6. Выполнение кода функции.

0xBA	IN ax, [bp + 4]	ax = c {2}
0xBC	IN bx, [bp + 6]	bx = d {5}
0xBE	ADD ax, bx	ax = ax + bx
0xC0	OUT [bp – 2], ax	e = ax {c + d}

	Addr	Value	Comments	sp	**0xE9**
	...			bp	**0xED**
	0xE7			PC	0xC2
sp	**0xE9**		фрейм функции f()		
	0xEB	7			
bp	**0xED**	0xFD адрес BP в main()			
	0xEF	0xB0 адрес возврата			
	0xF2	2	c = 2		
	0xF4	5	d = 5		
	0xF5	резерв для return_value			
	0xF7		фрейм функции main() (локальные переменные)		
	0xF9	5			
	0xFB	2			
	0xFD	адрес BP в os()			
	0xFF	адрес возврата в os()			

Обратите внимание, чтобы получить доступ к аргументам функции необходимо к `BP` прибавлять смещение, а для доступа к локальным переменным вычитать. Это логично, т. к. аргументы и локальные переменные хранятся по разные стороны от адреса, на который указывает значение в регистре `BP`. В результате выполнения представленного фрагмента кода ассемблера в переменной e (ячейке `0xEB`) оказывается число 7.

7. Сохранение возвращаемого значениям

Значение из переменной `e [bp – 2]` копируется через регистр ax в ячейку с адресом `0xF5` (`0xED{BP} – 8 = 0xF5`), зарезервированную под возвращаемое значение.

0xC2	IN ax, [bp – 2]	ax = e {7}
0xC4	OUT [bp + 8], ax	ret_val = ax {7}

	Addr	Value	Comments	sp	**0xE9**
	...			bp	**0xED**
	0xE7			PC	0xC6
sp	**0xE9**		фрейм функции f()		
	0xEB	7			
bp	**0xED**	0xFD адрес BP в main()			
	0xEF	0xB0 адрес возврата			
	0xF2	2	c = 2		
	0xF4	5	d = 5		
	0xF5	7	return_value		
	0xF7		фрейм функции main() (локальные переменные)		
	0xF9	5			
	0xFB	2			
	0xFD	адрес BP в os()			
	0xFF	адрес возврата в os()			

8. Эпилог функции.

Задача эпилога функции подготовить стек к возврату в вызывающую функцию.

Выполняется в два действия.

1. Очистка стека от локальных переменных. Этого можно добиться, спустив вершину стека на дно.

0xC6	MOV sp, bp

	Addr	Value	Comments	sp	0xED
	...			bp	**0xED**
	0xE7			PC	0xC8
	0xE9				
	0xEB				
sp,bp	0xED	0xFD адрес BP в main()			
	0xEF	0xB0 адрес возврата			
	0xF2	2	c = 2		
	0xF4	5	d = 5		
	0xF5	7	return_value		
	0xF7		фрейм функции main() (локальные переменные)		
	0xF9	5			
	0xFB	2			
	0xFD	адрес BP в os()			
	0xFF	адрес возврата в os()			

2. Восстановление дна стекового фрейма в состояние, до вызова функции `f()`. Для этого дастаточно извлечь значение из стека (регистр `SP` указывает на ячейку стека с адресом `0xED`) в регистр BP

0xC8	POP bp

	Addr	Value	Comments	sp	0xEF
	...			bp	0xFD
	0xE7			PC	**0xCA**
	0xE9				
	0xEB				
	0xED				
sp	0xEF	0xB0 адрес возврата			
	0xF2	2	c = 2		
	0xF4	5	d = 5		
	0xF5	7	return_value		
	0xF7		фрейм функции main() (локальные переменные)		
	0xF9	5			
	0xFB	2			
bp	0xFD	адрес BP в os()			
	0xFF	адрес возврата в os()			

9. Возврат в вызывающую функцию

0xCA	RET 4

	Addr	Value	Comments	sp	0xF5
	...			bp	0xFD
	0xE7			PC	0xB0
	0xE9				
	0xEB				
	0xED				
	0xEF				
	0xF2				
	0xF4				
sp	0xF5	7	return_value		
	0xF7		фрейм функции main() (локальные переменные)		
	0xF9	5			
	0xFB	2			
bp	0xFD	адрес BP в os()			
	0xFF	адрес возврата в os()			

Команда `RET` сперва извлечет из стека адрес возврата `0xB0` и запишет его в программный счетчик `PC`, при этом `SP` станет равным `0xF2`. Далее, из `SP` вычтется значение `4`, что приведет к очищению стека от аргументов функции `f()`. В итоге `SP` будет указывать на ячейку, в которой хранится значение, возвращенное функцией `f()`.

10. Извлечение из стека значения, возвращенного функцией

0xB0	POP ax
0xB2	OUT [bp - 6], ax

	Addr	Value	Comments	sp	0xF7
	...			bp	0xFD
	0xE7				
	0xE9				
	0xEB				
	0xED				
	0xEF				
	0xF2				
	0xF4				
	0xF5				
sp	0xF7	7	фрейм функции main() (локальные переменные)		
	0xF9	5			
	0xFB	2			
bp	0xFD	адрес BP в os()			
	0xFF	адрес возврата в os()			

Значение сохранилось в `res`. Состояние стека вернулось в первоначальное состояние, которое было до вызова функции `f()`.

Предупреждение: Имеется опечатка в коде ассемблера. После команды с адресом `0xB2` *должна следовать команда* `RET`*, которая завершит выполнение функции* `main()` *и передаст управление операционной системе. Данная ошибка не мешает усвоению основной идеи, и, поскольку, лень — двигатель прогресса, исправлять ее не будем.*

Теперь должно быть понятно, почему в следующем примере переменная `a` после вызова функции `inc(a)` не изменяется:

```
void inc(int a) {
        a = a + 1;
        return;
}
int main () {
        int a;
        a = 5;
        inc(a);
        std::cout << a; // 5
}
```

Переменные `a` располагаются в разных стековых фреймах. Под одну переменную `a` отведено место в стековом фрейме функции `main()`, под дргую в стековом фрейме функции `inc()`.

Программу следует изменить, чтобы функция `inc()` возвращала значение, а в функции `main()` это значение принималось:

```
int inc(int a) {
        a = a + 1;
        return a;
}
int main () {
        int a = 5;
        a = inc(a);
        std::cout << a; // 6
}
```

8.3 Статические переменные

Статическая переменная создаётся один раз и навсегда, вплоть до окончания работы программы. Обыкновенные локальные переменные располагаются в стеке и исчезают после завершения функции.

```
void test() {
	static int static_a = 0;
	int nostatic_a = 0;

	std::cout << nostatic_a << "\t" << static_a << std::endl;
	++static_a;
	++nostatic_a;
}

int main() {
	std::cout << "no_static\tstatic" << std::endl;
	for(int i = 0; i < 5; ++i) {
		test();
	}
}
```

Вывод:

```
no_static	static
0		0
0		1
0		2
0		3
0		4
```

Функция `test()` вызывается пять раз. При каждом вызове функции создаётся локальная переменная `nostatic_a`, которой присваивается `0`. Поэтому в первом столбце всегда выводится ноль. Статическая переменная `static_a` инициализируется только один раз, при первом вызове функции `test()` и больше не исчезает до конца программы. При каждом следующем вызове функции повторная инициализация не происходит, поэтому значения во втором столбце изменяются.

Убедимся, что статические переменные, хранятся в области оперативной памяти `.data`, а не в `User stack`. Определим функцию `test()`, в которой инициализируется статическая и нестатическая переменная и выводятся их адреса. Функция `test2()` необходима, чтобы проверить, что при помещении в стек переменной `nop`, адрес нестатической переменной изменится, а статической останется прежним:

```
void test() {
	static int static_a = 0;
	int no_static = 0;
	std::cout << &no_static << "\t" << &static_a << std::endl;
}

void test2() {
	int nop;
	test();
}

int main() {
	std::cout << "no_static \t static" << std::endl;
	test();
	int b = 5;
	test();
}
```

Адреса для нестатической и статической переменной принадлежат разным блокам памяти. Нестатическая переменная при повторных вызовах функции не изменяет свой адрес.

```
no_static	 static
0x...66d4	0x...e158
0x...66e4	0x...e158
```

Статическая переменная одна и та же для всех вызовов функции. Не статическая переменная при каждом вызове функции создаётся каждый раз новая.

Следующий код выдаст неожиданный результат:

```
int main() {
        std::cout << "no_static \t static" << std::endl;
        test(); // 1
        int b = 5;
        test(); // 2
}

no_static          static
0x...66d4          0x...e158
0x...66d4          0x...e158
```

Казалось бы между первым и вторым вызовом `test()` в стек была помещена переменная `b`, и адрес для `nostatic_a` должен измениться. Оказывается компилятор, выделил место в стеке для локальной переменной в начале функции `main()`, когда первый вызов `test()` ещё не был произведён.

8.4 Глобальные переменные

Переменные объявленные вне всяких функций являются **глобальными**. Доступ к глобальным переменным возможен внутри любой функции:

```
#include <iostream>
int today = 6;
void nextDay() {
        ++today;
}
int main() {
        nextDay();
        std::cout << today; // 7;
}
```

Существует проблема при работе с глобальными переменными:

```
int today = 6;
void whatADay() {
        std::cout << today << ":\t" << &today << std::endl;
}

int main() {
        int today = 7;
        whatADay(); // 6;
        std::cout << today << ":\t" << &today; // 7
}
```

Локальная переменная `today` перекрыла внутри функции `main()` глобальное имя `today`.

Внутри `main()` переменная `today` не имеет никакого отношения к глобальной переменной `today`:

```
6:      0x56...10
7:      0x7f ...04
```

Да и адреса у них из разных сегментов памяти, т. е. глобальные переменные хранятся не в `User stack`, а в `.data` или `.bss` областях, в зависимости от того, инициализированы они или нет.

Оператор разрешения области видимости `::` позволяет добраться до глобальной переменной, даже если локальная переменная имеет тоже самое имя:

```
int today = 6;
void whatADay() {
        std::cout << today << ":\t" << &today << std::endl;
}
int main() {
        int today = 7;
        whatADay(); // 6;
        std::cout << ::today; // 6
}
```

Лучший способ использования глобальной переменной — не объявлять её.

8.5 Указатели

В C++ имеются специальный тип, который позволяет хранить адреса на ячейки оперативной памяти, те самые 0xFB. Данный тип называется указатель.

8.5.1 Основные приёмы работы с указателями

Чтобы объявить указатель необходимо после названия типа поставить символ *:

```
int g = 5;
int* p = &a;
std::cout << a << std::endl;
std::cout << p << std::endl;
```

Переменная p теперь хранит не целое число, а адрес ячейки, в которой располагается переменная типа int. Указатель хранит адрес, взятие адреса записывается как &a, следовательно, можно записать выражение p = &a. Посмотрим на содержимое стека, после выполнения представленных строчек кода:

User stack		
	...	...
	0x0FFF3	мусор
p	0x0FFFB	0x0FFFF
a	0x0FFFF	5

Обратите внимание 0x0FFFB – 0x0FFF3 = 8, т. е. для хранения переменной типа указатель отводится 8 байт, а не четыре sizeof(p) = 8. Это связано с большим объёмом оперативной памяти. С использованием 4 байт, можно сохранить адрес одной ячейки из 4 миллиардов ячеек (2^{32}), что соответствует памяти не более 4 Гб. В современных компьютерах оперативная память больше, поэтому и места для хранения адреса нужно больше. На наших схемах стека для удобства адреса записываются короткими. Реальный вывод для переменной p выглядит следующим образом:

```
0x7fff38b9eeec
```

Если перевести данное число в десятичную систему счисления, получим:

```
140 734 145 097 452
```

Это значение явно больше четырёх миллиардов и не поместится в 4 байта. Указатель хранит адрес и занимает в памяти 8 байт.

Проведём эксперимент.

Программа:

```
char a_char;
int a_int;
double a_double;
char* p_char = &a_char;
int* p_int = &a_int;
double* p_double = &a_double;
std::cout << sizeof(a_char)         <<  " char "       << sizeof(p_char)          << std::endl;
std::cout << sizeof(a_int)          <<  " int "        << sizeof(p_char)          << std::endl;
std::cout << sizeof(a_double) << " double " << sizeof(p_double)     << std::endl;
```

вывод на экран:

```
1 char 8
4 int 8
8 double 8
```

Несмотря на то, что типы `char, int, double` занимают разное количество байт, указатель на переменную любого типа будет занимать в памяти 8 байт. Указатель хранит адрес первого байта, с которого начинается запись значения переменной в память. У переменной любого типа есть этот первый байт. И адрес этого первого байта весит одинаково для всех типов. Адресу вообще всё равно, что по нему хранится.

Указатели можно присваивать друг другу, если они имеют одинаковые типы.

```
int a = 5;
int* p1 = &a;
int* p2 = p1;
```

User stack		
p2	0x0FFF3	0x0FFFF
p1	0x0FFFB	0x0FFFF
a	0x0FFFF	5

Указатель p1 хранит адрес переменной a, указатель p2 также хранит адрес переменной a.

С указателями можно производить арифметические операции сложения и вычитания. Указатель это адрес, следовательно запись `p + 1` эквивалентна `p - sizeof(p_type)`. Относительно указателей не следует придумывать ничего волшебного. Если `p` имеет тип `int`, то запись `p + 1`, вычтет из значения указателя `sizeof(int) = 4`.

Пример кода:

```
int a = 5;
int* p1 = &a;
int* p2 = p1 + 1;
```

User stack		
	...	...
p2	0x0FFF3	0x0FFFB
p1	0x0FFFB	0x0FFFF
a	0x0FFFF	5

Теперь p2 указывает на ячейку 0x0FFFB, в которой хранится p1. Почему так лучше не делать, будет показано в следующих разделах.

> Примечание: если на вашей машине при добавлении переменных в стек их адреса увеличиваются, то `p + 1` приведёт к увеличению значения указателя на 4, прибавление сдвигает значение указателя вверх по стеку.

8.5.2 Оператор разыменования

Оператор * **разыменовывает** адрес. Термин не вполне удачный, т. к. никаких имён нет. На самом деле * позволяет по адресу получить доступ к ячейке.

Рассмотрим пример:

```
int a = 5;
std::cout << a << std::endl;  // 5
std::cout << &a << std::endl; // 0x0FFFF
std::cout << *&a << std::endl; // 5
```

User stack		
	...	...
	0x0FFF3	мусор
	0x0FFFB	мусор
a	0x0FFFF	5

- В первой строчке выводится значение из ячейки, куда была помещена переменная `a`.
- Во второй строчке выводится адрес этой ячейки.
- В третьей строчке с использованием `&a` сперва получаем адрес, а потом звёздочкой `*` разыменовываем его. Другими словами получаем доступ к ячейки по этому адресу, т. е. до значения переменной `a`.

Рассмотрим ещё несколько примеров:

```
int a = 5, b = 6, c = 7;
std::cout << *(&a + 2); // 7
std::cout << *(&a + 3); // ???
```

User stack		
	0x0FFF3	мусор
c	0x0FFF7	7
b	0x0FFFB	6
a	0x0FFFF	5

- В первой строчке получаем адрес `&a == 0x0FFFF`, прибавляем к нему два `&a + 2 == 0x0FFF7`, разыменовываем полученный адрес `*(0x0FFF7)` и получаем доступ к ячейке в которой хранится 7, т. е. к переменной `c`.
- Во второй строчке уже прибавляется 3, получается адрес `0x0FFF3`. В данной ячейке ещё не было ничего сохранено, поэтому на экране появится мусор.

Продемонстрируем ещё более удивительный пример:

```
int a = 5;
*(&a + 1) = 7;
int b;
cout << b; // 7
```

Здесь переменная `b` ничем не проинициализирована, но строчкой выше мы поместили значение 7 как раз в ту ячейку, в которой позднее будет объявлена переменная `b`.

Предупреждение: *Это просто учебные примеры, цель которых дать более глубокое представление об адресах и указателях. Разумеется при решении реальных задач не стоит устраивать такие головоломки. Тем более* ***компилятор не гарантирует****, что переменные в стеку будут располагаться именно в том порядке, который* ***мы ожидаем****. Код следует создавать простым и понятным для других.*

А как вам такой пример:

```
int a = 5;
{
        int c = 7;
}
int d = 8;
std::cout << c; // error: ‘c’ was not declared in this scope
std::cout << *(&a + 1); // 7
```

Здесь переменная `c` объявлена внутри `{ scope }`. За пределами `{...}` доступа к ней из программы нет. Однако значение в стеке продолжает хранится, пока не перезапишется другим числом.

Приведём пример, который показывает, что играть с адресами следует аккуратнее.

```
int     a = 5;
short c1 = 6;
short c2 = 7;
*(&a + 1) = 1234567890;
std::cout << sizeof(short) << std::endl; // 2
std::cout << c1 << std::endl; // 18838
std::cout << c2 << std::endl; // 722
```

Наверное вы ожидали увидеть `c1 = 1234567890`, `c2 = 7`. Как ни странно вывод `c1 = 18838`, `c2 = 722` очень даже логичный. Дело в том, что тип `short` занимает 2 байта, в том время как `int` 4 байта. Рассмотрим схему стека, в которой показаны все ячейки сразу после объявления переменных:

User stack		
Переменная	**Адрес**	**Содержимое**
	...	...
	0x0FFF7	мусор
c2	0x0FFF8	00000000
	0x0FFF9	00000011 (7)
c1	0x0FFFA	00000000
	0x0FFFB	00000110 (6)
a	0x0FFFC	00000000
	0x0FFFD	00000000
	0x0FFFE	00000000
	0x0FFFF	00000101 (6)

Выражение `(&a + 1)` даст адрес `0x0FFFB`, по которому начинает хранится `c1`. Переведём `1234567890` в 2ю систему счисления:

01001001100101100000001011010010

Для удобства разделим длинное число по восемь бит:

01001001 10010110 00000010 11010010

Данные четыре байта будут записаны, начиная с адреса `0x0FFFB`. Стек будет иметь следующее содержимое:

User stack		
Переменная	**Адрес**	**Содержимое**
	...	...
	0x0FFF7	мусор
c2	0x0FFF8	01001001
	0x0FFF9	10010110 (7)
c1	0x0FFFA	00000010
	0x0FFFB	11010010(6)
a	0x0FFFC	00000000
	0x0FFFD	00000000
	0x0FFFE	00000000
	0x0FFFF	00000101 (6)

Два младших байта ушли в переменную `c1`:

00000010 11010010 = 18838

Два старших байта ушли в переменную `c2`:

01001001 10010110 = 722

Вот и весь секрет.

На самом деле в результате запуска программа не выполнится до конца, а аварийно завершится с ошибкой:

```
*** stack smashing detected ***: terminated
Aborted (core dumped)
```

Следует отключить защиту компилятора, для таких странных экспериментов, использовать следующую команду для компиляции:

```
g++ main.cpp -fno-stack-protector
```

8.5.3 Разыменование указателей

Указатель хранит адрес, следовательно его можно разыменовать. Оператором * чаще всего разыменовывают именно указатели.

```
int a = 5;
int* p1 = &a;
int* p2 = p1;
*p2 = 7;
std::cout << a; // 7
std::cout << *p1; // 7
std::cout << *p2;// 7
```

User stack		
	...	...
p2	0x0FFF3	0x0FFFF
p1	0x0FFFB	0x0FFFF
a	0x0FFFF	5 → 7

В программе оба указателя `p1` и `p2` ссылаются на один и тот же адрес, по которому начинает храниться содержимое переменной `a`. Поэтому и вывод везде одинаковый.

8.5.4 Ещё раз об операторах

Следуйте следующим правилам при работе с указателями:

- Используйте амперсанд `&a`, когда необходимо узнать адрес, по которому хранится переменная.
- При объявлении указателя используйте `type*`.
- При разыменовании указателя `int* p;` используйте `*p`.

Путаница возникает из-за того, что `*` с одной стороны используется при объявлении указателя, т. е. делает из переменной место для хранения адреса. С другой стороны `*` выполняет противоположное действие — разыменовывает этот самый адрес, позволяя добраться до значения.

Запомните: `&a` - взятие адреса, `*p` - доступ к значению, а ещё `*` используют при объявлении указателей.

Следующий пример должен окончательно вас запутать

```
int a = 5;
int *p = &a;
std::cout << p; // 0x0FFFF
std::cout << &p; // 0x0FFFB
std::cout << *p; // 5
```

User stack		
	...	...
	0x0FFF3	мусор
p	0x0FFFB	0x0FFFF
a	0x0FFFF	5

В строке `std::cout << &p` программист запутался в `*&*&*` и вывел вместо значения переменной, на которую указывает указатель, адрес ячейки в которой хранится сам указатель.

8.6 Применение указателей

Разберём некоторые области применения указателей.

8.6.1 «Ссылка» на переменную

Указатели используются, чтобы сохранить адрес ячейки, в которую помещена переменная.

```
int a = 5;
int* p = &a;
```

Переменную p теперь можно применять в тех контекстах, где требуется адрес переменной, а не её значение. Например, при передаче в функцию.

8.6.2 Массивы в стеке и указатели

Рассмотрим, как в стеке располагается следующий массив:

```
const int N = 5;
int arr[N] = {1, 2, 3, 4, 5};
```

User stack		
	...	...
arr[4]	0x0FFEB	5
arr[3]	0x0FFEF	4
arr[2]	0x0FFF3	3
arr[1]	0x0FFF7	2
arr[0]	0x0FFFB	1
N	0x0FFFF	5

Элементы массива располагаются в стеке друг за другом. Каждый элемент занимает `sizeof(int) = 4 байта`.

Выведем на экран значение переменной `arr` без указания квадратных скобок и адрес по которому хранится нулевой элемент массива.

```
std::cout << arr << " ? " << &arr[0]; // 0x00FB ? 0X00FB
```

Оказывается что вывод для `arr`, совпадает с `&arr[0]`. Делаем вывод, что имя массива `arr`, является указателем на начало массива, т. е. на элемент с индексом ноль.

Изменить последний элемент теперь можно двумя способами:

*(arr + N - 1) = 7 или arr[N - 1] = 7.

Квадратные скобки, более удобная и понятная форма записи для обращения к элементам массива. Писать четыре символа `* ( + )` долго, кроме того не понятно, является ли тогда `arr` массивом. Квадратные же скобки имеются только у массива и у лямбда-функций, которые рассматриваются в следующих главах.

Кстати вывод `&arr` и `arr` выдают один и тот же адрес. Физически для переменной `arr` место в стеке отсутствует. Компилятор в процессе своей работы просто сгенерирует машинный код уже с нужными адресами. Если написать `*arr` или `arr[0]`, компилятор будет обращаться к ячейке `0x0FFFB + 0`.

Можно сохранить адрес на начало массива `0x0FFFB` в стеке, для этого следует записать значение `arr` в указатель:

```
int *p = arr; // обратите внимание, arr уже готовый адрес и пишется без амперсанда
std::cout << p; // 0x00FB
```

Следующий код выведет все элементы массива.

```
const int N = 5;
int arr[N] = {1, 2, 3, 4, 5};
int* p = arr;
for(int i = 0; i < N; ++i) {
        std::cout << *(p + i);
}
```

User stack		
	...	...
p	0x0FFE7	0x00FB
arr[4]	0x0FFEB	5
arr[3]	0x0FFEF	4
arr[2]	0x0FFF3	3
arr[1]	0x0FFF7	2
arr[0]	0x0FFFB	1
N	0x0FFFF	5

В этом примере можно обойтись и без переменной p, просто писать arr.

Представленный выше цикл выглядит не очень то указательным. Запишем его по другому:

```
for(int* it = arr; it != arr + N; ++it) {
        std::cout << *it;
}
```

Разберёмся, как происходит перебор элементов массива:

- Инициализирующее выражение `int* it = arr` создаёт указатель на первый элемент массива.
- Через этот указатель в теле цикла выводится значение элемента `std::cout << *it`.
- В конце каждой итерации указатель увеличивается на единицу `++it`. Увеличение на единицу указателя переводит его к ячейке, в которой хранится следующее значение. Такие трюки совершать безопасно, т. к. все элементы массива в стеке занимают одинаковое количество ячеек. Странная ситуация «размытия числа» по нескольким переменным, которая появилась в примере с short, исключена.
- Интересно выглядит условие продолжения цикла `it != arr + N`. Последний элемент имеет адрес `&arr[N - 1]`, который проще записывается `arr + N - 1`. Последняя итерация должна выполниться при `it == arr + N - 1 == 0x0FFEB`. Цикл следует завершить, когда it уйдёт за последний элемент. Адресом «запоследнего» элемента является `arr + N == 0x00E7`. Итак, условие `"пока it не достигло конца массива"` выглядит следующим образом `it != arr + N`.

Представленный пример описывает идею итераторов. **Итератором** является некоторый объект, который позволяет обойти по порядку все элементы некоторой последовательности. Последовательностью, в рассмотренном примере, является массив, а итератором указатель it. Увеличение it на единицу осуществляет переход к следующему элементу последовательности. Продолжать обход следует пока итератор it не уйдёт за пределы последовательности (массива). Несуществующий элемент, расположенный по адресу `arr + N`, находится за пределами последовательности и имеет научное название `past - the - last - element`.

Итераторы широко используются в C++, положены в основу стандартной библиотеки STL и общей парадигмы программирования, в том числе и в других языках, например, Python.

8.6.3 Передача в функцию

Заставим функцию inc() изменять значение переменной. Используем в аргументе указатель:

```
void inc(int* p) {
    *p = *p + 1; // изменяем значение, хранящееся по адресу p
}
int main () {
    int a = 5;
    inc(&a);
    std::cout << a; // 6
}
```

Обратите внимание, что в функции inc() теперь следует передавать не переменную (т. е. целое число), а адрес переменной. Запись inc(&a) наводит на мысль, что внутри inc() переменная a может измениться, происходит передача адреса ячейки, в которой хранится a. Этот адрес можно разыменовать *p и изменить значение ячейки, отведённой под хранение переменной a.

Массив является указателем, следовательно, он естественным образом передаётся в функцию через указатель. Вывод массива на экран является частой задачей. Напишем функцию, выводящую массив на экран, которая будет принимать указатель и количество элементов:

```
void pretty_print(int* arr, int N) {
    for(int i = 0; i < N; ++i) {
        std::cout << arr[i] << ", ";
    }
}
int main() {
    const int N = 5;
    int arr[N] = {1, 2, 3, 4, 5};
    pretty_print(arr, N);
}
```

Посмотрим, как может выглядеть функция, которая находит минимальный и максимальный элементы массива:

```
void minmax(int* arr, int N, int* min, int* max) {
	*min = *max = arr[0];
	for(int* it = arr; it != arr + N; ++it ) {
		if(*it < *min) {
			*min = *it;
		}
		if(*it > *max) {
			*max = *it;
		}
	}
}
int main() {
	const int N = 5;
	int arr[N] = {1, 2, 3, 4, 5};
	int max, min;
	minmax(arr, N, &min, &max);
	std::cout << min << max; // 1  5
}
```

Массивы передаются по адресу, следовательно функция может изменять значения в элементах массива, объявленного в функции `main()`:

```
void x2(int* arr, int N) {
	for(int* it = arr; it != arr + N; ++it) {
		*it *= 2;
	}
}
int main() {
	const int N = 5;
	int arr[N] = {1, 2, 3, 4, 5};
	x2(arr, N);
	pretty_print(arr, N); // 2, 4, 6, 8, 10
}
```

Указатели удобно использовать, при передачи переменной в функцию если:

- Функции требуется изменить передаваемую переменную.
- Из функции требуется вернуть несколько значений.
- Передаваемая переменная объявлена динамически.

8.7 Динамическое выделение памяти

Попробуем создать массив из двух миллионов элементов.

```
int main() {
    int arr[2'000'000];
}
```

Программа успешно скомпилируется и выполнится до конца.

Увеличим количество элементов на сто тысяч:

```
int main() {
    int arr[2'100'000];
}
```

Программа успешно скомпилируется, но при выполнении возникнет ошибка сегментации:

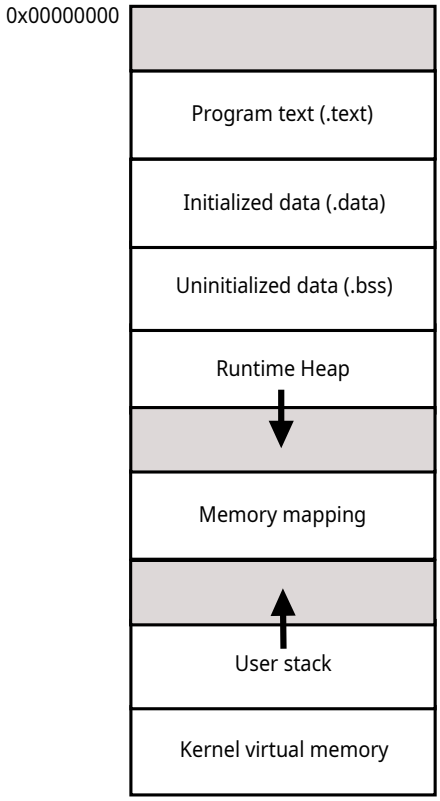
```
Segmentation fault (core dumped)
```

8.7.1 Куча

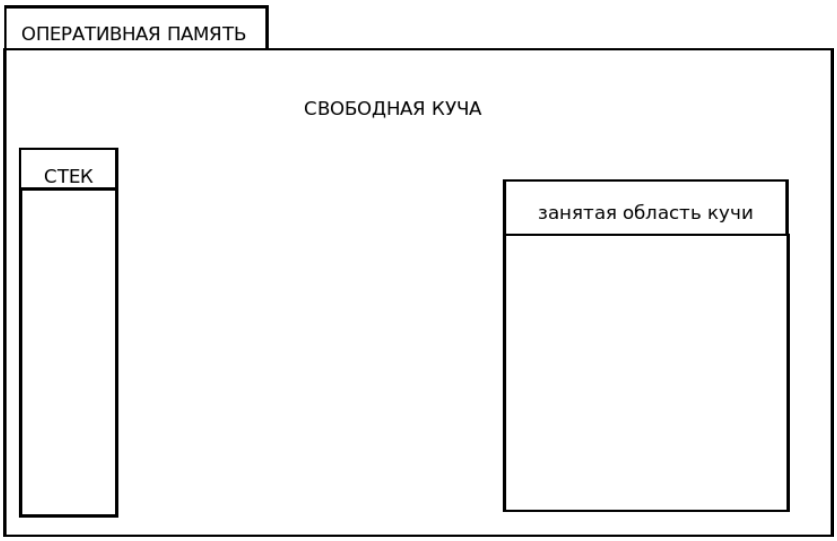
Ошибка сегментации возникает, когда программа пытается обратиться к области памяти на которую у неё нет доступа. Все дело в том, что стек вызовов `User Stack` не слишком большая область в памяти. Оценку объёма стека можно получить, вычислив сколько памяти занимает массив из двух миллионов элементов типа `int`:

```
2млн. X 4 байта = 8 млн. байт ≈ 8 Мбайт.
```

В нашем случае получилось чуть более восьми мегабайт. Для хранения больших массивов данных используется другая область оперативной памяти — **Runtime heap или куча**.



Воспользуемся упрощённой моделью. Под кучей будем понимать — всю свободную область оперативной памяти. Часть кучи занята данными самой операционной системы, процессами других программ, «висящих» в данный момент в оперативной памяти. Запущенные процессы можно наблюдать в диспетчере задач.



Стек предназначен для хранения текущих «лёгких» переменных, таких как числа, небольшие массивы и структуры. Данные, занимающие много места следует располагать в куче. В стеке сохраняют адрес на область в куче, куда были помещены данные.

8.7.2 Ключевое слово new

Разместим целое число типа `int` в куче. Для этого объявим переменную указатель `int* p`. Данный указатель способен хранить адрес ячейки, в которой находится число типа `int`. Ключевое слово `new` позволяет выделить память в куче. Если после слова `new` написать `int`, в куче будет выделено 4 байта для хранения одного целого числа. Оператор `new` возвращает адрес на начало блока выделенной памяти, т. е. адрес на первую ячейку блока.

Исследуем, какими будут адреса у переменных при сохранении их в стеке и куче:

```
int main() {
    int a = 5; // значение 5 запишется в стек
    int* p = new int; // в стек сохраниться адрес на ячейку из кучи
    *p = 6; // поместится в кучу
    int b = 7; // запишется в стек
    std::cout << &a << &p << &b; // 0x000FF    0x000FB    0x000F3
    std::cout << p; //  0x555AB
    delete p;
}
```

User stack		
	...	...
c	0x0FFF3	7
p	0x0FFFB	0x555AB
a	0x0FFFF	5

Runtime heap	
...	...
0x555A7	мусор
0x555AB	**6**
0x555AF	мусор

Переменные `a`, `p` и `b` сохраняются в стеке, их адреса изменяются последовательно. Ключевое слово `new` выделило ячейку с адресом `0x555AB`, которая находится в совсем другой области памяти, нежели стек. Записан этот адрес был в переменную указатель. Обратиться к динамически выделенному блоку в куче можно разыменовав указатель: `*p = 6`.

8.7.3 Ключевое слово delete

Скомпилируем и запустим следующий код:

```
void intToHeap(int x) {
	int* p = new int(x);
}

int main() {
	for(int i = 0; i < 1'000'000'000; ++i) {
		intToHeap(5);
	}
}
```

Компьютер начнёт жутко виснуть и если повезёт через несколько десятков секунд получим сообщение:

```
Killed
```

Процесс с нашей программой был уничтожен операционной системой, поскольку он занял всю возможную оперативную память. Функция `intToHeap()` размещает в куче целое число. Вызов функции в цикле производится миллиард раз, для успешной работы программы требуется 4Гб свободной оперативной памяти:

```
1 млрд X 4 байта = 4 млрд. байт ≈ 4 Гб
```

После завершения функции очищается только стек, поэтому его переполнения не происходило. Память, выделенную оператором `new` необходимо очищать после использования, т. е. делать её свободной.

Удаление блока памяти из кучи выполняет оператор `delete`:

```
void intToHeap(int x) {
	int* p = new int(x); // выделяем область в куче
	*p += 5; // используем выделенную область
	delete p; // удаляем область из кучи
}
```

Если есть `new`, обязательно должен быть `delete`.

Никогда не освобождайте уже освобождённую область:

```
void intToHeap(int x) {
	int* p = new int(x); // выделяем область в куче
	*p += 5; // используем выделенную область
	delete p; // удаляем область из кучи
	delete p; // повторное освобождение
}
```

Данный код может привести к непредсказуемым результатам.

Также никогда не используйте область после её освобождения:

```
void intToHeap(int x) {
	int* p = new int(x); // выделяем область в куче
	*p += 5; // используем выделенную область
	delete p; // удаляем область из кучи
	*p += 6; // непредсказуемое поведение
}
```

8.7.4 Размещение массива в куче

Ключевое слово new позволяет разместить массив в куче. Массив по прежнему остаётся непрерывным блоком ячеек с фиксированным размером. Однако, теперь компилятору знать размер массива не обязательно. Количество элементов может быть получено в процессе выполнения программы, например, командой `cin`:

```
int main() {
	int N;
	std::cin >> N;
	int* arr = new int[N];
	for(int* it = arr; it != arr + N; ++it) {
		std::cout << *it;
	}
	delete[] arr;
}
```

Массив после использования следует удалить, т. к. при его создании было использовано ключевое слово `new`, т.е. он был размещён в куче. Обратите внимание, что для удаления массива используется оператор `delete[]` с квадратными скобками.

- Массивы следует удалять `delete[]` с квадратными скобками.
- Не массивы следует удалять `delete` без квадратных скобок.
- Это важно.

Массивы, размещённые в куче, часто называют **динамическими**. В некоторых контекстах, под динамическими массивами понимают массивы с переменной длинной. В C++ массивы всегда имеют фиксированную длину, которая определяется в момент их создания. Слово динамический, в данном случае, говорит о динамическом выделении памяти в куче, а не о подвижности размеров массива.

На ЭВМ с оперативной памятью 8 Гб удалось разместить массив из 2.5 млрд элементов:

```
int main() {
        int* arr = new int[2'500'000'000];
}
```

Напомним, что при запуске программы операционная система создаёт для неё виртуальную оперативную память. Часть этой памяти операционная система на своё усмотрение может размещать на жёстком диске, а не в оперативной памяти. Также операционная система может перемещать части виртуальной памяти с жёсткого диска в реальную оперативную память и обратно, чтобы сохранить быстродействие работы всей системы в целом.

Программист, к счастью, имеет доступ только к виртуальной памяти, и не обязан постоянно думать о реальной физической памяти... Если он, конечно, сам не пишет код операционной системы…

Размещение массива из 2.6 млрд элементов привело к краху программы с ошибкой:

```
terminate called after throwing an instance of 'std::bad_alloc'
  what():  std::bad_alloc
Aborted (core dumped)
```

8.8 Ссылки

Ссылки представляют собой псевдоним для переменной. Сами ссылки не хранятся ни в стеке, ни в куче. Ссылка физически не существует. Ссылка — высокоуровневая конструкция для написания эффективного кода. Особенно ссылки полезны при передачи в функцию.

8.8.1 Создание ссылки

Ссылка создаётся подобно объявлении переменной, только после типа пишут знак амперсанда &.

```
int a = 5;
int& b = a;
++b;
std::cout << a << b; // 6  6
```

User stack		
	...	...
	0x0FFFB	
a	0x0FFFF	5

Обратите внимание, что в стек не добавляется никаких новых переменных.

Строкой `int& b = a` в программе объявляется для ячейки `0x0FFFF` новое имя `b`. Нет никакой разницы между `a` и `b`. Оба имени связаны с одной и той же ячейкой памяти.

Ссылки не существует физически, поэтому при объявлении её необходимо инициализировать. Если вы попытаетесь объявить переменную, не присвоив ей никакого значения:

```
int a = 5;
int& b;
```

Получите ошибку:

```
error: ‘b’ declared as reference but not initialized
```

Переводится ошибка следующим образом:

```
ошибка: ‘b’  объявления как ссылка, но не проинициализирована.
```

Создать ссылку можно только при её объявлении:

```
int a = 5, c = 7;
int& b = a; // создание ссылки на переменную a
b = c; // через ссылку b в переменную a записывается значение 7 из c
c = 9;
std::cout << b; // 7
```

Запись `b = c` обыкновенное присваивание. Не существует синтаксиса, который позволил бы для переменной `b` присвоить другу ссылку. Ссылке вообще ничего нельзя присвоить, т. к. её не существует. Ссылку можно только проинициализировать один раз при её создании:

```
int& b = a;
```

Не стоит путать ссылки с указателями:

- **Указатель** — конкретная переменная, хранящая адрес, его можно разыменовывать и изменять
- **Ссылка** — псевдоним, второе имя, синоним для другой переменной.

8.8.2 Передача в функцию по ссылке

Бессмысленно в функции `main()` создавать синонимы для переменной. Приведёт это разве только к запутанности программного кода. Основная область применение ссылок — передача аргументов в функцию.

Вспомним пример с инкрементом:

```
int inc(int x) {
    x = x + 1;
    return x;
}
int main() {
    int a = 5;
    a = inc(a); // 6
    a = inc(7); // 8
}
```

При вызове функции `a = inc(a)` происходит следующее:

- Объявляется переменная `x`.
- Значение из переменной `a` копируется в `x`.
- Увеличивается значение переменной `x`.
- Значение копируется из `x` обратно в `a` после возврата из функции.

При вызове `a = inc(7)` происходит примерно тоже самое.

Перепишем код с использованием ссылки в качестве аргумента:

```
void inc(int& x) {
	x = x + 1;
}
int main() {
	int a = 5;
	inc(a); // 6
	inc(7); // error
}
```

При вызове `inc(a)` к переменной `a` прибавляется единица, т. к. теперь `x` это ссылка или второе имя для переменной `a`, а не отдельная ячейка в стеке.

Вызов `inc(7)` приведёт к ошибке:

```
error: cannot bind non-const lvalue reference of type ‘int&’ to an rvalue of type ‘int’
```

Ошибка гласит, что аргумент ссылка `int& x` ожидает получить `lvalue` (т. е. ячейку в стеке), но получает вместо этого `rvalue` значение типа `int`.

Ссылка - синоним для какой-то области в памяти. Области, для которой можно узнать адрес, грубо говоря, называются **`lvalue`**. Обычно в качестве `lvalue` выступают переменные. Все для чего нельзя узнать адрес, это **`rvalue`**. Невозможно узнать по какому адресу хранится число 7, поэтому это `rvalue`.

В таком контексте ссылки лучше не использовать. Допустим у нас имеется функция проверки чётности числа `isEven()`, которая принимает число по ссылке. Что вы ожидаете увидеть после выполнения данного кода:

```
int main () {
	int a = 8;
	if(isEven(a)) {
		std::cout << a; // ?? 8 ??
	}
}
```

Разумеется число 8. Но не тут-то было. Т. к. переменная передаётся по ссылке, то функция вполне может изменить в ней значение, не смотря на безобидное название:

```
bool isEven(int& x) {
	bool is_even = (x % 2 == 0); // определяем чётность
	if(is_even) {
		--x; // делаем подлянку
	}
	return is_even;
}
int main () {
	int a = 8;
	if(isEven(a)) {
		std::cout << a; // 7
	}
}
```

Изначально `a` равно `8`. Это число чётное, условие в `if()` срабатывает, но вывод озадачивает. Разумеется, никто специально не будет портить свои функции. Но в `C++` очень легко «`выстрелить себе в ногу`», т. е. ошибиться на ровном месте, упустив некоторую деталь.

8.8.3 Константность

Предполагается, что никакая функция по умолчанию не должна изменять передаваемые в неё переменные. В случае, с передачей по ссылке, такого поведения позволяет достичь понятие **константность**. Добавим ключевое слово `const` перед `int&`:

```
bool isEven(const int& x) {
	bool is_even = (x % 2 == 0);
	if(is_even) {
		--x; // error: decrement of read-only reference ‘x’
	}
	return is_even;
}
```

Ошибка возникает в тот самый момент, когда производится попытка изменить ссылку. Ссылка константная, а константные объекты изменять нельзя. Придётся убрать из функции подлянку:

```
bool isEven(const int& x) {
	return (x % 2 == 0);
}
```

Функция `isEven()` вызывается быстро, т. к. не происходит копирования в переменную `x`, при этом он такая же безопасная для вызывающей стороны как и `bool isEven(int x)`. Она гарантированно не изменит передаваемую переменную.

Используйте именно константные ссылки. Если функция должна изменить значение передаваемой переменной, лучше воспользоваться указателем:

```
bool isEven(int* x);
if(isEven(&a)) {
	std::cout << a;
}
```

Амперсанд в выражении `isEven(&a)` подсказывает, что-то тут не чисто. Функция принимает адрес переменной, а значит значение переменной `a` внутри функции скорее всего изменится.

8.8.4 Константные указатели

Указатели также можно объявлять константными. Константной может быть переменная на которую ссылается указатель, либо адрес, хранимый в указателе, либо и переменная и адрес. Различают три вида константности для указателей:

1. **Указатель на константу**. Невозможно изменить значение в ячейке, адрес которой хранится в указателе, однако можно присвоить указателю адрес другой переменной. Указатель изменяем, но то, на что он указывает константа:

```
int a = 7, b = 5;
const int* p = &a;
*p = 5; // error: assignment of read-only location '* p'
p = &b; // ok
```

2. **Константный указатель**. Противоположная ситуация. По адресу на которую указывает указатель изменения возможны, однако нельзя изменить адрес, хранимый в самой переменной p:

```
int a = 7, b = 5;
int* const p = &a;
*p = 5; // ok
p = &b; // error: assignment of read-only variable 'p'
```

3. **Константный указатель на константу**. Ни сам адрес, хранимый в указателе, ни ячейку на которую этот адрес указывает изменить через переменную p не получится.

```
int a = 7, b = 5;
const int* const p = &a;
*p = 5; // error: assignment of read-only location '*(const int*)p'
p = &b; // error: assignment of read-only variable 'p'
a = 8; // ok
std::cout << *p; // 8 ok
```

Сама переменная a при этом не является константой, поэтому её изменять можно. Просто через *p изменить не выйдет.

Чтобы не запутаться где писать `*`, пишите её всегда справа от типа. Перемещается ключевое слово `const`, а не оператор `*`. Существует тип `int` и «инт со звездочкой» `int*`. Не надо ничего придумывать относительно звёздочки.

Глава 9. Высокий уровень

Было рассмотрено, как устроено программирование на низком уровне. Люди придумали множество конструкций и идей, позволяющих ускорить и упростить процесс написания программ. Современные высокоуровневые архитектуры гибкие, позволяют изменять программы с минимальными финансовыми и временными затратами, делать их надёжными и устойчивыми к сбоям и ошибкам.

В данной главе рассматриваются следующие высокоуровневые идеи:

- Структуры
- Перегрузка операторов
- Шаблоны

Разработаем, в качестве примера, программу поиска победителя в олимпиаде.

Информация об участнике олимпиады описывается следующей структурой данных:

- Индивидуальный идентификационный номер.
- Возраст
- Первичный бал.

Итоговый балл рассчитывается по формуле `Первичный балл / Возраст`. Сравниваются итоговые баллы. Чем старше участник, тем больший первичный балл ему необходимо набрать, чтобы победить в олимпиаде.

Напишем программу, для хранения сведений об одном участнике:

```
double totalGrade(int grade, int age) {
    return static_cast<double>(grade) / age;
}

int main() {
    int id = 1;
    int age = 16;
    int grade = 97;
}
```

В функции вычисления итогового балла `totalGrade()` деление целочисленных типов приводит к целому числу, но итоговый бал может оказаться дробным. Приведение целого типа к дробному `static_cast<double>(grade)` преобразует целое число `grade` в дробное. Если хотя бы один делитель дробный, то результатом будет дробное число.

Информация об участниках оказалась разбросанной по трём переменным, а хотелось бы все сведения хранить в одном месте.

Попробуем сохранить информацию обо всех `N` участниках, придётся использовать массивы:

```
const int N = 5;
int id[N] = {1, 2, 3, 4};
int age[N] = {16, 15, 12, 17, 13};
int grade[N] = {32, 43, 32, 55, 45, 65};
```

Сведения об участниках оказались разбросанными по трём массивам, а хотелось бы всю информацию хранить в одном массиве.

9.1 Struct

Используемые до сих пор типы `int, double, bool, short, char` и т.д. являются базовыми типами языка `C++`. Программы без переменных данных типов трудно представить. Программистам также предоставляется возможность создавать свои собственные типы. Такие типы называются **структурами**. Структуры включают в себя несколько переменных, которые называются **полями структуры**.

9.1.1 Определение структуры

Структура для нашей задачи может выглядеть следующим образом:

```
struct Participant{
	int id;
	int age;
	int grade;
};
int main() {
	Participant pupil;
	pupil.id = 1;
	pupil.age = 16;
	pupil.grade = 1452;
	std::cout << totalGrade(pupil.grade, pupil.age);
}
```

Описание структуры начинается с ключевого слова `struct`, после которого следует название структуры. В фигурных скобках перечисляются поля. После закрывающейся скобки обязательно следует точка с запятой. Поставить её там с первого раза невозможно, т. к. обычно после `}` люди привыкли не ставить `;`. После объявления структуры `Participant` можно создавать переменные типа `Participant`, подобно как это делается с типом `int`.

Получить доступ к полям структуры можно через оператор точка:

```
pupil.id = 1;
```

Структуру можно разместить в куче, для этого объявляется указатель на структуру, а память под поля структуры выделяется оператором `new`:

```
Participant* pupil = new Participant;
```

Доступ к полю тогда потребует предварительного разыменования:

```
(*pupil).id = 1;
```

Автоматически разыменовывать указатель позволяет оператор `->`:

```
pupil->id = 1;
```

Правило простое:

- структура размещена в стеке, для доступа полей используйте точку,
- структура размещена в куче (хранится через указатель), используйте стрелочку вправо `->`.

Структура в памяти хранится единым блоком, поля структуры записываются друг за другом, подобно элементам массива:

```
sizeof(Participant) == 12 байт, 4 байт X 3
```

Массив хранит набор однотипных элементов, структура хранит признаки - поля, описываемые некоторый объект или явления. Структура тип данных, более сложный и интересный, чем `int`.

Значения для полей структуры при её создании можно задать через инициализатор. Значения для полей требуется передавать в том порядке, в котором поля объявлены в структуре.

```
Participant pupil = { 1, 16, 52 };
```

Инициализатор удобно использовать при определении массива типа `Participant`:

```
int main() {
    const int N = 5;
    Participant pupil[N] = { {1, 16, 1432},
                             {2, 15, 2543},
                             {3, 12, 5332},
                             {4, 17, 4345},
                             {5, 13, 3465} };
    std::cout << totalGrade(pupil[0].grade, pupil[0].age);
}
```

Внешние `{ }` являются инициализатором, в котором через запятую перечисляются элементы массива. Каждый элемент массива создаётся через инициализатор структуры `{ }`, в которой перечисляются поля структуры.

Функция `totalGrade()` рассчитывает итоговый балл для конкретного участника и является такой же неотъемлемой частью `Pariticipant` как и поля `id`, `age`, `grade`. Функции заключённые внутрь структуры называются методами:

```
struct Participant{
	int id;
	int age;
	int grade;
	double totalGrade() {
		return static_cast<double>(grade) / age;
	}
};
```

Расчёт итогового бала теперь выглядит максимально лаконично:

```
std::cout << pupil[0].totalGrade();
```

Размер структуры не изменился, несмотря на добавление метода:

```
Sizeof(Participant) == 12
```

Поля структуры относятся к данным и сохраняются в стеке вместе с обыкновенными переменными. Методы относятся к коду программы и хранятся в другом месте. Код метода сохранится один раз, вне зависимости от того, сколько переменных типа `Participant` будет объявлено. Поля же сохраняются в стеке для каждой переменной типа `Participant`.

9.1.2 Выравнивание в памяти

Рассмотрим другую структуру:

```
struct D {
	bool a;
	int b;
};
```

Размеры полей `sizeof(bool) == 1` и `sizeof(int) == 4`, должно получиться пять, но размер структуры оказывается восемь `sizeof(D) == 8`. Компилятором применяется выравнивание, в результате чего размер структуры оказывается кратен четырём. Для представленного кода:

```
struct D {
        bool a;
        int b;
};
int main() {
        int one;
        D d;
        int two;
        std::cout << &a << std::endl;
        std::cout << &d.a << std::endl;
        std::cout << &d.b << std::endl;
        std::cout << &b << std::endl;
}
```

содержимое стека оказывается следующим:

User Stack			
	Переменная	**Адрес**	**Содержимое**
Переменная типа D	**{int} d.b**	**0x0FF08**	00000000
		0x0FF09	00000000
		0x0FF0A	00000000
		0x0FF0B	00000000
	empty	**0x0FF0C**	
		0x0FF0D	
	{bool} d.a	**0x0FF0E**	00000000
Простые переменные	two	0x0FF0F	мусор
		0x0FF10	мусор
		0x0FF11	мусор
		0x0FF12	мусор
	one	0x0FF13	мусор
		0x0FF14	мусор
		0x0FF15	мусор
		0x0FF16	мусор

Обратите внимание, что переменные помещены в стек не в том порядке, в котором они объявлялись. Компилятор сперва разместил переменные `one` и `two`, а затем переменную типа структуры `D`. В самой же структуре сперва располагается переменная `a`, которая занимает 1 байт, после 3 байта пустоты и далее переменная `b`.

Поэтому странные трюки, которые рассматривались в разделе указатели, когда мы увеличивали значение указателя, разыменовывали его и тем самым изменяли значения других переменных, лучше не делать.

Также, обратите внимание, что поля структуры автоматически проинициализировались нулём, чего не скажешь о простых переменных, в которых остался лежать мусор.

9.1.3 Конструкторы

Для структур можно определить специальный метод – конструктор, который инициализирует поля структуры.

Конструктор – метод, который вызывается при инициализации переменной (создании объекта). Имя данного метода обязано совпадать с названием структуры. Тип возвращаемого значения не указывается. Конструкторов может быть несколько, если они отличаются набором аргументов:

```
struct Participant{
	int id;
	char age;
	short grade;
	Participant(int id) {
		this->id = id;
		age = 14;
		grade = 0;
	}

	Participant(int id, int age, int grade) {
		this->id = id;
		this->age = age;
		this->grade = grade;
	}
	double totalGrade() {
		return static_cast<double>(grade) / age;
	}
```

```
    void show() {
        std::cout << "{ id: " << id << ", age: " << age << ", grade: " << grade << "}";
    }
};
```

Ключевое слово `this` это указатель на текущий объект, поэтому для доступа к полям текущего объекта используется разыменование через стрелочку:

```
this->id = id;
```

Писать `this` в данном примере обязательно, т. к. `id` будет обозначать аргумент, который передаётся в конструктор, а не поле объекта. Если такой неоднозначности нет, то `this` можно не писать.

Создать участника следующими способами:

```
Participant pupil1(1, 16, 56);
Participant pupil2(5);
Participant pupil3;

pupil1.show(); // { id: 1, age: 16, grade: 56 };
pupil2.show(); // { id: 5, age: 14, grade: 0 };
pupil3.show(); // { id: 0, age: 0, grade: 0 };
```

Напишем структуру, которая внешне будет выглядеть, как массив переменной длины:

```
#include <iostream>
#include <cassert>

struct DynamicArr {
    int capacity = 1000;
    int size;
    int* arr;
    DynamicArr(int size) {
        assert(size < capacity);
        arr = new int[capacity];
        this->size = size;
    }
```

```
        void push_back(int element) {
                assert(size < capacity);
                arr[size] = element;
                ++size;
        }
        void pop_back() {
                assert(size > 0);
                --size;
        }
};
```

В структуре имеется три поля:

- `capacity` — фактический размер реального массива (ёмкость),
- `size` – количество элементов в динамическом массиве,
- `arr` – указатель на массив, размещённый в куче.

Обратите внимание на конструктор. Инициализатор `{ }` не позволил бы сконструировать столь сложный объект. В конструкторе происходит следующее:

1. Макрос `assert()` пишет сообщение об ошибке и завершает программу в том случае, если выражение в круглых скобках ложно. Если условие `size < capacity` не выполняется, то программа завершит выполнение с ошибкой. Это предостерегает структуру от переполнения реального массива `arr`, который имеет длину `capacity`. Для использования `assert()` следует подключить библиотеку `<cassert>`
2. Оператором `new` выделяется место под реальный массив в куче размером `max_size`.
3. Полю `this->size` присваивается значение, переданное через аргумент `size`.

Наличие поля `size` позволяет обращаться с массивом `arr` фиксированной длины, будто у него длина может изменяться.

Метод `push_back()` добавляет элемент к массиву, виртуальный размер `size` увеличивается.

Метод `pop_back()`, уменьшает `size` на единицу, тем самым как бы удаляется последний элемент.

Пример использования данной структуры:

```
void useArr() {
	DynamicArr da(0);
	da.push_back(1);
	da.push_back(2);
	da.push_back(3);
	for(int i = 0; i < da.size; ++i) {
		std::cout << da.arr[i]; // 1 2 3
	}
}

int main() {
	useArr();
}
```

Функция `main()` вызывает функцию `useArr()`, в которой создаётся переменная `da` типа `DynamicArr`. Изначально массив da считается пустым (`size = 0`), хотя фактически в памяти уже выделено место для 1000 элементов. Далее, последовательно добавляются элементы. В итоге размер `size` становится равным трём. Элементы выводятся на экран. Функция `useArr()` завершается. Выполнение программы возвращается в `main()`. Стек очищается, переменная `da` оказывается больше недоступной, однако в куче остался массив из 1000 элементов.

Мы забыли освободить место, выделенное в куче оператором `new` для массива `arr`. В конце функции `useArr()`, следует очистить память:

```
	delete[] da.arr;
```

В функции `useArr()` нет ключевого слова `new`, поэтому совсем не очевидно, что в ней следует освобождать память с использованием `delete[]`. Преодолеть данное затруднение могут деструкторы.

9.1.4 Деструктор

Деструктор — метод, который вызывается при уничтожении объекта. Деструктор начинается с символа тильда ~, далее следует название структуры:

```
struct DynamicArr {
	// ..
	~DynamicArr() {
		delete[] arr;
	}
	//..
};
```

В отличии от конструктора, деструктор можно определить только один. Нельзя придумать несколько способов для уничтожения объекта. Тем более список аргументов у деструктора всегда пуст, т. е. невозможно создать перегруженный деструктор, отличающийся набором аргументов.

Добавим в конструктор и деструктор команды вывода на экран, чтобы понять в какой момент создаётся (вызывается конструктор) и уничтожается (вызывается деструктор) объект da.

Полный код примера:

```
#include <iostream>
#include <cassert>

struct DynamicArr {
	int capacity = 1000;
	int size;
	int* arr;
	DynamicArr(int size) {
		assert(size < capacity);
		std::cout << "Array created" << std::endl;
		arr = new int[capacity];
		this->size = size;
	}
```

```
        ~DynamicArr() {
                std::cout << "Array destroyed" << std::endl;
                delete[] arr;
        }
        void push_back(int element) {
                assert(size < capacity);
                arr[size] = element;
                ++size;
        }
        void pop_back() {
                assert(size > 0);
                --size;
        }
};

void useArr() {
        DynamicArr da(0);
        da.push_back(1);
        da.push_back(2);
        da.push_back(3);
        for(int i = 0; i < da.size; ++i) {
                std::cout << da.arr[i]; // 1 2 3
        }
        std::cout << std::endl;
}


int main() {
        std::cout << "Start" << std::endl;
        useArr();
        std::cout << "End" << std::endl;
}
```

Из выводимой информации:

```
Start
Array created
123
Array destroyed
End
```

видно, что объект `da` создаётся в начале функции `useArr()`, а уничтожается сразу после завершения функции `useArr()`, в момент когда очищается стек.

9.1.5 RAII

Способ управления ресурсами в куче, описанный в рассмотренном примере, называется RAII (*Resource Acquisition Is Initialization*) — получение ресурса есть инициализации. Способ заключается в следующем:

1. В конструкторе оператором `new` выделяется ресурс в куче.

2. В деструкторе оператором `delete` (для массивов `delete[]`) память, занятая ресурсом, освобождается.

Ресурс в куче находится пока объект (переменная), которому принадлежит ресурс, существует.

9.2 Перегрузка операторов

Ознакомьтесь со следующим кодом.

```
#include <iostream>
#include <cassert>

struct DynamicArr {
	int capacity = 1000;
	int size;
	int* arr;
	DynamicArr(int size) {
		assert(size < capacity);
		arr = new int[capacity];
		this->size = size;
	}
	~DynamicArr() {
		delete[] arr;
	}
	void push_back(int element) {
		assert(size < capacity);
		arr[size] = element;
		++size;
	}
	void pop_back() {
		assert(size > 0);
		--size;
	}
};

void fillByOrder(DynamicArr& a, int start = 0) {
	for(int i = 0; i < a.size; ++i) {
		a.arr[i] = i + start;
	}
}
```

```
void showArr(const DynamicArr& a) {
	bool isFirst = true;
	std::cout << "[ ";
	for(int i = 0; i < a.size; ++i) {
		if(isFirst) {
			isFirst = false;
		}
		else {
			std::cout << ", ";
		}
		std::cout << a.arr[i];
	}
	std::cout << " ]" << std::endl;
}

int max(int a, int b) {
	if(a > b) {
		return a;
	}
	return b;
}

DynamicArr sumArr(const DynamicArr& a1, const DynamicArr& a2) {
	DynamicArr res(max(a1.size, a2.size));
	for(int i = 0; i < res.size; ++i) {
		res.arr[i] = a1.arr[i] + a2.arr[i];
	}
	return res;
}
```

```
int main() {
    DynamicArr arr1(5);
    DynamicArr arr2(4);
    fillByOrder(arr1); // { 0, 1, 2, 3, 4 }
    fillByOrder(arr2, 10); // {10, 11, 12, 13}
    DynamicArr arr3 = sumArr(arr1, arr2);
    showArr(arr3); // [10, 12, 14, 16, 4]
}
```

В представленном коде происходит следующее:

- Создаётся два массива `arr1` и `arr2` типа `DynamicArr`.
- Функция `fillByOrder(DynamicArr& a, int start = 1)` заполняет переданный по ссылке динамический массив значениями по порядку, начиная со `start`. Если при вызове функции будет передан только один аргумент — массив, то для `start` будет задано значение по умолчанию `start = 1`. Массив передаётся по ссылке, чтобы изменения происходили в самом массиве, а не в его копии, которая уничтожится после вызова функции.
- Функция `sumArr(const DynamicArr& a1, const DynamicArr& a2)` возвращает новый массив, каждый элемент которого равен сумме соответствующих элементов `a1` и `a2`. Размер результирующего массива равен максимальному размеру из `a1` и `a2`. Обратите внимание, массивы передаются по ссылке, чтобы избежать копирования. Ссылка сделана константной, чтобы избежать случайного изменения исходных массивов. Более того, имеется возможность передавать переменные, объявленные с ключевым словом `const`:

```
const DynamicArr arr1;
const DynamicArr arr2;
sumArr(arr1, arr2); // ok, если бы не было const, была бы error
```

- Функция `showArr(const DynamicArr& a)` выводит элементы массива в красивых квадратных скобках. После последнего элемента запятая не выводится. Для этого в функции использовался трюк с флагом `isFirst`. Массив также передаётся по константной ссылке, чтобы избежать копирования тяжеловесного объекта типа `DynamicArr`.
- Вспомогательная функция `int max(int a, int b)` возвращает максимальное значение переменной из `a` и `b`.

Заменим следующую строку кода:

```
DynamicArr arr3 = sumArr(arr1, arr2);
```

на более дружелюбную запись:

```
DynamicArr arr3 = arr1 + arr2; // error
```

Получим следующую ошибку:

```
error: no match for ‘operator+’ (operand types are ‘DynamicArr’ and ‘DynamicArr’)
```

Дословно переводится:

```
отсутствует соответствие для оператора+ (типы операндов DynamicArr и DynamicArr)
```

Другими словами, действие для оператора `+` над переменными типа `DynamicArr` не задано. Данное действие можно задать следующим образом:

```
//..
DynamicArr operator+(DynamicArr& left, DynamicArr& right) {
    return sumArr(left, right);
}
int main() {
    //..
    DynamicArr arr3 =  arr1 + arr2; // ok
}
```

Оператор плюс представляет собой обычную функцию, принимающую два аргумента и возвращающую значение некоторого типа. Многие языки программирования высокого уровня позволяют задавать действия для операторов - **перегружать операторы**. Таким

образом, был перегружен оператор `+`, который принимает два операнда типа `DynamicArr` и возвращает значение типа `DynamicArr`.

В принципе, можно перегрузить `operator+` много раз и заставить его складывать помидоры с огурцами и получать банку консервированных овощей.

Посмотрим на запись:

```
std::cout << a;
```

Оператор `operator<<`, также как и `operator+`, обладает левым операндом (поток вывода типа `std::ostream`) и правым операндом (выводимое значение). Перегрузим `operator<<` таким образом, чтобы он выводил массив типа `DynamicArr`:

```
std::ostream& operator<<(std::ostream& os, const DynamicArr& a) {
    bool isFirst = true;
    os << "[";
    for(int i = 0; i < a.size; ++i) {
        if(isFirst) {
            isFirst = false;
        }
        else {
            os << ", ";
        }
        os << a.arr[i];
    }
    os << "]" << std::endl;
    return os;
}
int main() {
    //..
    DynamicArr arr3 =  arr1 + arr2;
    std::cout << arr3; // [11, 13, 15, 17, 5]
}
```

Функция `operator<<` возвращает ссылку на тот же самый поток вывода, что позволяет осуществлять следующие цепочки вывода:

```
std::cout << one << two << std::endl << "Hello";
```

Аналогичным образом можно перегрузить любой оператор:

```
>, <, <=, >=, ==, !, !=, -, *, /, &, &&, |, || и т.д.
```

9.3 Копирование объектов

Рассмотрим следующий пример:

```
int main() {
    DynamicArr a1(5);
    fillByOrder(a1); // 1, 2, 3, 4, 5
    DynamicArr a2;
    a2 = a1;
    a2.arr[0] = 7;
    std::cout << a1; // [7, 2, 3, 4, 5]
}
```

В выражении `DynamicArr a2 = a1` происходит копирование полей из `a1` в `a2`. В переменную `a2.arr` копируется значение указателя из поля `a1.arr`. Указатели `a1.arr` и `a2.arr` указывают на один и тот же массив в куче. Поэтому, при изменении значения `a2.arr[0]` изменяется и `a1.arr[0]`. Однако, ожидалось получить два разных массива.

Более того, при завершении программы произошла ошибка:

```
double free or corruption (!prev)
Aborted (core dumped)
```

После завершении функции `main()` произошло уничтожение объекта `a2`, для чего вызвался его деструктор `a2.~DynamicArr()`, в котором произошло освобождение памяти для массива `delete[] arr`. Далее уничтожается объект `a1.~DynamicArr()` в котором повторно освобождается та же самая область памяти, что может привести к непредсказуемо катастрофическим последствиям. Компилятор сумел распознать «выстрел себе в ногу» и приложение аварийно завершилось, не успев наделать много бед.

Данную оплошность позволит исправить перегрузка оператора присваивания. Её следует определить внутри `struct DynamicArr`. Оператор будет принимать один аргумент, ссылку на переменную справа от `=`, а возвращать ссылку на текущий объект `*this`:

```
struct DynamicArr {
	// ...
	DynamicArr& operator=(const DynamicArr& rhs) {
		if(this == &rhs) { // если присваиваем самому себе
		return  *this; // просто возвращаем себя
		}
		size = rhs.size;
		for(int i = 0; i < size; ++i) {
			arr[i] = rhs.arr[i]; // копируем значения из правой переменной
					// вместо копирования указателя arr = rhs.arr
		}
		return *this;
	}
};
```

Теперь в куче два разных массива, код работает как нужно. Однако, запись `DynamicArr a2 = a1` вновь приведёт к той же самой неприятной ситуации:

Дело в том, что в данном случае `a2` ещё не создан и срабатывает не оператор присваивания, а, так называемый, **конструктор копирования**, который по умолчанию просто копирует указатели. Его можно переопределить следующим образом:

```
struct DynamicArr {
	// ...
	DynamicArr(const DynamicArr& rhs) {
		arr = new int[capacity];
		size = rhs.size;
		for(int i = 0; i < size; ++i) {
			arr[i] = rhs.arr[i];
		}
	}
};
```

Перегрузка операторов сравнения пригодится позже. Полный код программы выглядит следующим образом:

```
#include <iostream>
#include <cassert>

struct DynamicArr {
	int capacity = 1000;
	int size;
	int* arr;
	DynamicArr(int size) {
		assert(size < capacity);
		arr = new int[capacity];
		this->size = size;
	}
	~DynamicArr() {
		delete[] arr;
	}
	void push_back(int element) {
		assert(size < capacity);
		arr[size] = element;
		++size;
	}
	void pop_back() {
		assert(size > 0);
		--size;
	}
	DynamicArr(const DynamicArr& rhs) {
		arr = new int[capacity];
		size = rhs.size;
	for(int i = 0; i < size; ++i) {
			arr[i] = rhs.arr[i];
		}
	}
```

```
        DynamicArr& operator=(const DynamicArr& rhs) {
                if(this == &rhs) {
                        return  *this;
                }
                size = rhs.size;
                for(int i = 0; i < size; ++i) {
                        arr[i] = rhs.arr[i];
                }
                return *this;
        }
};

void fillByOrder(DynamicArr& a, int start = 1) {
        for(int i = 0; i < a.size; ++i) {
                a.arr[i] = i + start;
        }
}
int max(int a, int b) {
        if(a > b) {
                return a;
        }
        return b;
}
DynamicArr sumArr(const DynamicArr& a1, const DynamicArr& a2) {
        DynamicArr res(max(a1.size,a2.size));
        for(int i = 0; i < res.size; ++i) {
                res.arr[i] = a1.arr[i] + a2.arr[i];
        }
        return res;
}
DynamicArr operator+(DynamicArr& left, DynamicArr& right) {
        return sumArr(left, right);
}
```

```
std::ostream& operator<<(std::ostream& os, const DynamicArr& a) {
	bool isFirst = true;
	os << "[";
	for(int i = 0; i < a.size; ++i) {
		if(isFirst) {
			isFirst = false;
		}
		else {
			os << ", ";
		}
		os << a.arr[i];
	}
	os << "]" << std::endl;
	return os;
}

int main() {
	DynamicArr arr1(5);
	fillByOrder(arr1); // 1, 2, 3, 4, 5
	DynamicArr arr2 = arr1;
	arr2.arr[0] = 7;
	std::cout << arr1;
}
```

9.4 Перегрузка функций

Взглянем на функцию, которая вычисляет среднее арифметическое элементов массива типа `DynamicArr`:

```
double average(const DynamicArr& a) {
	int sum = 0;
	for(int i = 0; i < a.size; ++i) {
		sum += a.arr[i];
	}
	return static_cast<double>(sum) / a.size;
}
```

Можно написать функцию, которая делает тоже самое для обычного массива типа `int[]`:

```
double average(int* a, int size) {
	int sum = 0;
	for(int i = 0; i < size; ++i) {
		sum += a[i];
	}
	return static_cast<double>(sum) / size;
}
```

Следующая программа успешно скомпилируется и заработает:

```
int main() {
	DynamicArr arr(5);
	fillByOrder(arr);
	std::cout << average(arr) << std::endl;
	int a[5] = {5, 6, 7, 8, 9};
	std::cout << average(a, 5) << std::endl;
}
```

Оказывается можно объявлять функции с одинаковыми именами. Главное, чтобы функции отличались сигнатурами. Сигнатура включает в себя имя функции и список аргументов. Функции со следующими сигнатурами могут сосуществовать:

```
int func(int, int);
int func(int);
int func(double);
```

Следующие две функции отличаются только типом возвращаемого значения, который к сигнатуре не относится, поэтому придётся оставить только одну функцию:

```
int func(int)
void func(int)
```

В примере из прошлого раздела можно перегрузить функцию `max` для типа `int` и `double`:

```
int max(int a, int b) {
	if(a > b) {
	return a;
	}
	return b;
}
double max(double a, double b) {
	if(a > b) {
	return a;
	}
	return b;
}
```

Функции с одинаковыми именами, но разными сигнатурами, называют **перегруженными**.

Операторы также являются функциями и их можно перегружать. Например, сигнатуры функций с именами `operator+` отличаются типами левого и правого операндов:

```
Tomato operator+(const Apple& app1, const Apple& app2);
DynamicArr operator+(const DynamicArr&  app1, const DynamicArr&  app2);
```

Перегруженные функции никак не связаны между собой, не смотря на то, что имеют одинаковое имя. Действия которые выполняют функции определяются программистом. Следует писать код так, что бы функции с одинаковыми именами выполняли более менее

одинаковые действия. Это относится и к операторам. Не следует перегружать `operator*` таким образом, чтобы он выполнял вычитание левого операнда из правого.

9.5 Обобщённое программирование

Обобщённое программирование — это парадигма или способ написания программ, без указания конкретных типов. Пишется шаблонная программа, из которой компилируется конкретный код для конкретных типов.

9.5.1 Шаблонные функции

Обратите внимание, что реализации перегруженных функций `int max(int, int)` и `double max(double, double)` совпадают. Алгоритм один и тот же, отличие только в типах аргументов. Язык C++ позволяет записать данный алгоритм один раз, реализовав шаблонную функцию. Чтобы функцию сделать шаблонной, необходимо перед типом возвращаемого значения, дописать конструкцию `template<typename T>`. Теперь в функции можно использовать имя `T` для обозначения типа:

```
template<typename T> T max(T a, T b) {
	if(a < b) {
		return b;
	}
	return a;
}
```

Вызов шаблонной функции происходит следующим образом:

```
int main() {
	double a = 5.3, b = 7.5;
	max<double> (a, b); // 7.5
	max<int>(5, 7); // 7
}
```

Конкретной функции `max()` после определения ещё не существует. Шаблонная функция просто описывает алгоритм, физически она не компилируется в машинные коды. Реальный код функции компилятор сгенерирует, когда доберётся до строки, где осуществляется её вызов с конкретным типом, указанным в треугольных скобках. В представленном примере,

компилятор создаст код для двух различных функций. Одна будет принимать и возвращать тип `double`, другая тип `int`. В первом случае, вместо `T` подставится конкретный тип `double`, во втором случае `int`.

Если функция `max<double>()` ни разу не будет записана в программе, то компилятор не создаст машинный код для неё. А если она будет использована несколько раз, то компилятор сгенерирует код для функции `max<double>()` один раз. Будет происходить вызов этого кода, каждый раз когда программа вызывает `max<double>()`.

Конкретный тип в треугольных скобках при вызове обобщённой функции можно не указывать, если компилятор в состоянии сам его выявить, исходя из типов передаваемых аргументов:

```
int main() {
	double a = 5.3, b = 7.5;
	max (a, b); // 7.5 max<double>()
	max (5, 7); // 7 max<int>()
}
```

Вспомним структуру `Participant`:

```
struct Participant{
	int id;
	char age;
	short grade;
	double totalGrade() const {
		return static_cast<double>(grade) / age;
	}
};
```

Компилирование следующей программы приведёт к ошибке:

```
int main() {
	Participant p1 = {1, 7, 64};
	Participant p2 = {2, 16,  85};
	Participant win = max<Participant>(p1, p2);
	std::cout << win.id; // 1
}
```

```
error: no match for ‘operator<’ (operand types are ‘Participant’ and ‘Participant’)
  56 |       if(a < b) {
```

Похожая ошибка имела место быть, при попытке сложения двух массивов без перегрузки `operator+`. Даже с точки зрения логики, не понятно как правильно сравнивать участников. Сравнивать можно по возрасту, количеству набранных баллов или `id`. Перегрузим `operator<` для типа `Participant`, чтобы сравнение происходило по итоговому баллу:

```
bool operator<(const Participant& lhs, const Participant& rhs) {
    return lhs.totalGrade() < rhs.totalGrade();
}
```

Код заработает и на экране появится `id` победителя, равный 1, т. к. `(64 / 7) > (85 / 16)`.

Обратите внимание, что в объявление метода `double totalGrade() const {}` в структуре `Participant` в конце добавилось ключевое слово `const`. Данное слово, гарантирует, что внутри метода `totalGrade() const` поля структуры не будут изменены. После этого можно вызывать `totalGrade()` для константных объектов, таких как аргументы `lhs` или `rhs` функции `operator<()`. Удалите `const` из объявления `double totalGrade() {}` и получите ошибку.

Для полного счастья перегрузим оставшиеся операторы сравнения. Трюк в том, что достаточно написать реализацию только для `operator<` и `operator==`, оставшиеся операторы будут производными:

```
bool operator<(const Participant& lhs, const Participant& rhs) {
    return lhs.totalGrade() < rhs.totalGrade();
}
bool operator==(const Participant& lhs, const Participant& rhs) {
    return (lhs.totalGrade() - rhs.totalGrade()) * (lhs.totalGrade() - rhs.totalGrade()) < 1e-10;
}
bool operator!=(const Participant& lhs, const Participant& rhs) {
    return !(lhs == rhs);
}
bool operator<(const Participant& lhs, const Participant& rhs) {
    return rhs < lhs;
}
```

```
bool operator>=(const Participant& lhs, const Participant& rhs) {
        return !(lhs<rhs);
}
bool operator<=(const Participant& lhs, const Participant& rhs) {          return !(lhs>=rhs);  }
```

9.5.2 Шаблонные структуры

Шаблонными можно делать не только отдельные функции, ни и целые структур. Ранее была разработана достаточно полезная структура `DynamicArr`, позволяющая создавать динамический массив. У неё имеется существенный недостаток, можно создавать только массивы типа `int`. Сделаем структуру обобщённой:

```
template <typename T>
struct DynamicArr<T> {
        //...
        T* arr;
        DynamicArr(int size) {
                // ...
                arr = new T[capacity];
                // ...
        }
        DynamicArr(const DynamicArr& rhs) {
                arr = new T[capacity];
                // ...
        void push_back(T element) {
                //...
};
```

Всего лишь потребовалось в четырёх местах заменить `int` на имя `T`, обозначающее некоторый любой тип.

Создадим динамический массивы целых чисел, дробных чисел, и даже динамический массив участников типа `Particiapnt`:

```
int main(){
    DynamicArr<int> i_arr(0);
    i_arr.push_back(7);
    DynamicArr<double> d_arr(0);
    d_arr.push_back(7.5);
    DynamicArr<Participant> p_arr(0);
    d_arr.push_back({1, 7, 64});
}
```

Правда придётся скорректировать сигнатуру для `fillByOrder()`, т. к. она работает только с целочисленными массивами:

```
void fillByOrder(DynamicArr<int>& a, int start = 1) { ... }
```

И сделать функции `sumArr()`, `operator+()`, `operator<<()`, `average()` обобщёнными:

```
template <typename T>
DynamicArr<T> sumArr(const DynamicArr<T>& a1, const DynamicArr<T>& a2) {...}

template <typename T>
DynamicArr<T> operator+(DynamicArr<T>& left, DynamicArr<T>& right) {...}

template <typename T>
std::ostream& operator<<(std::ostream& os, const DynamicArr<T>& a) {...}

template <typename T>
double average(const DynamicArr<T>& a) {...}
```

Добавим ещё одну полезную обобщённую функцию `swap()`, которая обменивает содержимое переменных местами:

```
template <typename T> void swap(T& a, T& b) {
    T temp = a;
    a = b;
    b = temp;
}
```

Аргументы передаются по ссылке, чтобы изменялись значения в самих передаваемых переменных, а не в их локальных копиях, создаваемых в функции `swap()`:

```
int main() {
    int a = 5, b = 6;
    swap(a, b);
    std::cout << a << b; // 6 5
    swap(7, 8);         // error: cannot bind non-const lvalue reference of type 'int&' to an rvalue
                        // of type 'int'
}
```

Ошибка во втором вызове `swap()` логична, т. к. передаются не переменные, а числа. Функция `swap()` ожидает именно ссылку на переменную, т. е. `lvalue`.

Создадим два массива участников за первый и второй день, соответственно. Функция `swap()` справится и с динамическим массивом:

```
int main() {
    DynamicArr<Participant> first_day(1000);
    DynamicArr<Participant> second_day(1000);
    swap(first_day, second_day);
}
```

Рассмотрим подробнее, как будет происходить обмен значений в переменных `first_day` и `second_day`. Проанализировав вызов `swap(first_day, second_day)` компилятор поймёт, что требуется реализовать шаблонную функцию `swap<DynamicArr<Participant>>` и скомпилирует код для следующей функции:

```
void swap(DynamicArr<Participant>& a, DynamicArr<Participant>& b) {
    DynamicArr<Participant> temp = a;
    a = b;
    b = temp;
}
```

В первой строчке функции при инициализации переменной `temp` вызывается копирующий конструктор:

```
DynamicArr(const DynamicArr& rhs) {
        arr = new T[capacity];
      size = rhs.size;
      for(int i = 0; i < size; ++i) {
                  arr[i] = rhs.arr[i];
       }
  }
```

В куче создаётся массив `arr` из 1000 элементов, происходит копирование размера из переменной `a` и всех 100 значений в цикле в созданный массив `arr`.

В строке `a = b` вызывается перегруженная операция присваивания, где опять таки копируется 100 элементов из `b.arr` в `a.arr`

В строке `b = temp` снова выполняется копирование 100 элементов из `temp.arr` в `b.arr`;

В момент завершения функции `swap()` вызывается деструктор `temp.~DynamicArr()`, память выделенная под хранение `temp.arr` освобождается.

Произошло копирование более трёхста значений, а достаточно бы было просто поменять значения указателей `b.arr` и `p.arr` местами. Для обмена переменных типа `DynamicArr<T>` лучше разработать специальную функцию:

```
template <typename T>
void swapDynamicArr(DynamicArr<T>& a, DynamicArr<T>& b) {
        swap(a.size, b.size);
        swap(a.arr, b.arr);
}
```

И никаких копирований больших блоков памяти. Представьте, сколько бы времени было сэкономлено, если бы массивы содержали миллионы элементов. Функция `swap()` универсальная. Всё, что универсально, не всегда работает достаточно быстро. Об этом следует помнить, ведь язык C++ один из самых быстрых. Неправильно на таком языке писать неэффективные программы.

Шаблонные функции и структуры мощное средство, которое позволяет реализовать парадигму обобщённого программирования.

9.5.3 Итераторы

Напишем функцию `maxElement()`, которая находит максимальный элемент массива `DynamicArr<T>`:

```
template <typename T>
T maxElement(const DynamicArr<T>& a) {
	assert(a.size > 0);
	T max = a.arr[0];
	for(int i = 1; i < a.size; ++i) {
		if(a.arr[i] > max) {
			max = a.arr[i];
		}
	}
	return max;
}

int main() {
	DynamicArr<Participant> p(0);
	p.push_back({1, 16, 54});
	p.push_back({2, 16, 78});
	p.push_back({3, 16, 46});
	Participant win = maxElement(p);
	std::cout << win.id;
}
```

Алгоритм поиска максимального значения последовательности очень популярен. Функция `maxElement()` может работать только со структурами типа `DynamicArr<T>`. Перепишем функцию так, чтобы она могла искать максимальный элемент в любой последовательности, хоть в обычном массиве `int arr[5]`. Функция будет принимать итераторы на начало и на конец последовательности, а возвращать указатель на максимальный элемент:

```
template <typename T>
T maxElement(T begin, T end) {
	assert(end - begin > 0); // последовательность не пустая
	T max = begin;
	for(T it = begin; it != end; ++it) {
		if(*it > *max) {
			max = it;
		}
	}
	return max;
}
```

Тип `T` выступает в роли итератора. Итератор - некоторый тип, который содержит информацию о текущем элементе. Инкремент переменной типа `T`, должен привести итератор к следующему элементу. Итератор `begin` — указывает на первый элемент последовательности, итератор `end` — на конец последовательности, точнее на первый элемент за последним элементом. В простейшем случае итератором является указатель на элемент массива. Для целочисленного массива все просто, сама переменная является указателем на первый элемент, т. е. это `begin`. Если к указателю на первый элемент прибавить количество элементов, то уйдём за пределы массива, это и есть `end`:

```
int main() {
	DynamicArr<Participant> p(0);
	p.push_back({1, 16, 54});
	p.push_back({2, 16, 78});
	p.push_back({3, 16, 46});
	const int N = 5;
	int arr[N] = { 1, 2, 7, 5, 4};
	int* max = maxElement(arr, arr + N);
	std::cout << *max; // 7
}
```

Для `DynamicArr` вызов будет следующим:

```
int main() {
	DynamicArr<Participant> p(0);
	p.push_back({1, 16, 54});
	p.push_back({2, 16, 78});
	p.push_back({3, 16, 46});

	Participant* win = maxElement(p.arr, p.arr + p.size);
	// либо, что тоже самое
	// Participant* max = maxElement(&p.arr[0], &p.arr[p.size]);
	std::cout << win->id; // 2
}
```

Можем пойти дальше, и добавить в структуру `DynamicArr<T>` два метода, которые будут возвращать итераторы на начало и конец массива, т. е. указатели на первый и на первый за последним элемент:

```
template <typename T>
struct DynamicArr {
	// ...
	T* begin() {
		return &arr[0];
	}
	T* end() {
		return &arr[size];
	}
};
```

Поиск победителя теперь выглядит понятнее:

```
Participant* win = maxElement(p.begin(), p.end());
```

Разыменование указателя для структуры доступно через оператор `->`, т. е. вместо `(*win).id`, можно написать `win->id`.

9.6 Статические функции

Прежнюю функцию `double average(const DynamicArr<T>& a)` можно сделать методом структуры `DynamicArr<T>`. Функцию `DynamicArr<T> sumArr(const DynamicArr<T>& a1, const DynamicArr<T>& a2)` лучше сделать статической функцией внутри структуры `DynamicArr<T>`:

```
template <typename T> struct DynamicArr {
	// ...
	double average() {
		int sum = 0;
		for(int i = 0; i < size; ++i) {
			sum += arr[i];
		}
		return static_cast<double>(sum) / size;
	}
	static DynamicArr sumArr(const DynamicArr& a1, const DynamicArr& a2) {
		DynamicArr res(max(a1.size, a2.size));
		for(int i = 0; i < res.size; ++i) {
			res.arr[i] = a1[i] + a2[i];
		}
		return res;
	}
};
```

Вызов метода `average()` будет осуществляться через переменную типа `DynamicArr`:

```
DynamicArr<int> arr(5);
arr.average();
```

Для вызова статической функции структуры переменная не нужна. Достаточно просто написать двоеточие после имени структуры и далее название статической функции. Тогда в перегрузку `operator+` для динамического массива следует добавить `DynamicArr<T>::`

```
template <typename T>
DynamicArr<T> operator+(DynamicArr<T>& left, DynamicArr<T>& right) {
    return DynamicArr<T>::sumArr(left, right);
}
```

Обыкновенные нестатические методы производят действия с полями текущего объекта, например, метод `average()` вычисляет среднее арифметическое массива `this->arr`для текущего объекта.

Статическая функция отличается от обыкновенного нестатического метода, тем, что для её вызова не нужен объект. Следовательно, статическая функция не имеет доступа к полям текущего объекта. Внутри статической функции отсутствует указатель на текущий объект `this`. Обычно статическими делают вспомогательные функции, которые логически относятся к структуре, но не нуждаются в объекте для своей работы. Функция `sumArr(,)` суммирует два массива, которые ей передаются в качестве аргументов. Сама же она не принадлежит никакому объекту.

Полный дополненный вариант структуры `DynamicArr<T>` будет представлен в конце следующей главы.

9.7 Передача функции в качестве аргумента

Допустим, хотим сравнить, кто из двух участников старше. Функция `maxElement()` здесь не поможет, т. к. участники сравниваются по итоговому баллу (см. перегрузку операторов сравнения для структуры `Participant`), а не возрасту. Напишем перегрузку для функции `maxElement()`, которая принимает третьим параметром компаратор. **Компаратор** — функция, которая указывается как сравнивать элементы. Функция принимает два аргумента и возвращает `true`, если первый аргумент меньше второго. Так принято, что компаратор реализует операцию меньше.

9.7.1 Передача функции через указатель

Отвлечёмся, и посмотрим как передать в функцию в качестве аргумента другую функцию. Для примера разберём программу, которая выводит таблицу `x | f(x)` на экран:

```
double parabola(double x) {
	return x * x;
}
double hyperbola(double x) {
	return 1 / x;
}
void table (double start, double end, int N, double (*f)(double)) {
	double step = (end - start) / N;
	for(double x = start; x < end; x += step) {
		std::cout << x << "\t" << f(x) << std::endl;
	}
}
int main() {
	std::cout << "===parabola===" << std::endl;
	table(1, 10, 5, parabola);
	std::cout << "===hyperbola===" << std::endl;
	table(1, 10, 5, hyperbola);
}
```

Получится следующий вывод:

```
===parabola===
1	1
2.8	7.84
4.6	21.16
6.4	40.96
8.2	67.24
===hyperbola===
1	1
2.8	0.357143
4.6	0.217391
6.4	0.15625
8.2	0.121951
```

Перегруженный вариант функции `maxElement()` и компаратор для сравнения участников по возрасту выглядят следующим образом:

```
template <typename T>
T maxElement(T begin, T end, bool (*compare)(const T, const T)) {
	assert(end - begin > 0); // последовательность не пустая
	T max = begin;
	for(T it = begin; it != end; ++it) {
		if(compare(max, it)) { // if(max < it)
			max = it;
		}
	}
	return max;
}
bool CompareByAge(Participant* lhs, Participant* rhs) {
	return lhs.age < rhs.age;
}
```

9.7.2 Применение шаблонов

Сигнатуру для функции `maxElement()` можно упростить, заменив сложный тип третьего аргумента `bool (*comparator)(const T, const T)` на обобщенный тип, определив в шаблоне вторым аргументом тип `Comparator`, помимо типа `T`:

```
template <typename T, typename Comparator>
T maxElement(T begin, T end, Comparator compare) { ... }
```

Итого получается следующий пример:

```
template <typename T>
T maxElement(T begin, T end) {
	assert(end - begin > 0); // последовательность не пустая
	T max = begin;
	for(T it = begin; it != end; ++it) {
		if(*it > *max) {
			max = it;
		}
	}
	return max;
}
template <typename T, typename Comparator>
T maxElement(T begin, T end, Comparator compare) {
	assert(end - begin > 0); // последовательность не пустая
	T max = begin;
	for(T it = begin; it != end; ++it) {
		if(compare(max, it)) { // if(max < it)
			max = it;
		}
	}
	return max;
}

bool CompareByAge(Participant* lhs, Participant* rhs) {
	return lhs->age < rhs->age;
}
```

```
int main() {
    DynamicArr<Participant> p(0);
    p.push_back({1, 16, 100});
    p.push_back({2, 25, 100});
    p.push_back({3, 18, 100});

    Participant* win = maxElement(p.begin(), p.end());
    std::cout << win->id; // 3

    Participant* oldest = maxElement(p.begin(), p.end(), CompareByAge);
    std::cout << oldes->age; // 25
}
```

Сперва определяется победитель, для этого вызывается перегрузка `maxElement()` без компаратора, которая использует при сравнении `operator<`, перегруженный для типа `Participant`:

```
bool operator<(const Participant& lhs, const Participant& rhs) {
    return lhs.totalGrade() < rhs.totalGrade();
}
```

Для определения самого старшего участника вызывается версия `maxElement()`, которой третьим аргументом передаётся функция компаратор `CompareByAge`, сравнивающая участников по возрасту.

9.8 Лямбда-функции

Можно обойтись и без `CompareByAge()`, а использовать лямбда-функцию.

9.8.1 Использование лямбда-функций

Лямбда-функция — функция не имеющая имени, код которой пишется в том самом месте, где происходит её вызов:

```
Participant* oldest = maxElement(p.begin(),  p.end(), [](Participant* a, Participant* b) {
                                                                    return a->age < b->age });
```

Вместо имени функции записываются пустые квадратные скобки, а далее всё как в обычной функции `CompareByAge()`. Везде где компилятор может выявить тип из контекста, допускает писать `auto`, вместо названия типа. Ключевое слово `auto` особенно удобно применять в лямбда-функциях. Часто типы аргументов в лябда-функциях имеют длинное название, которое сложно выявить, а компилятор справится с этим без проблем:

```
Participant* oldest = maxElement(p.begin(),  p.end(), [](auto a, auto b) { return a->age < b->age });
```

Даже на одной строчке всё уместилось.

Попробуйте аналогично разработать две перегрузки функции `sort()`, которая осуществляет сортировку методом пузырька, разобранном в разделе массивы:

```
template <typename T>
void sort(T begin, T end) { ... } // используется < для сравнения
template <typename T, typename Comparator>
void sort(T begin, T end, Comparator compare) { ... }  // используется компаратор
```

9.8.2 Захват локальных переменных

Имеется возможность в лямбда-функцию передавать локальные переменные из вызывающей программы. Для этого в квадратных скобках пишется имя **захватываемой** переменной. Если предполагается, что лямбда функция может изменить переменную, то захват следует осуществлять по ссылке. Например, следующий код, помимо определения самого старшего участника, подсчитывает количество участников, старше 18 лет:

```
int main() {
	DynamicArr<Participant> p(0);
	p.push_back({1, 16, 100});
	p.push_back({2, 25, 100});
	p.push_back({3, 18, 100});
	int cnt = 0;
	Participant* oldest = maxElement(p.begin(), p.end(), [&cnt](auto a, auto b) {
					if(b->age >= 18) {
						++cnt; // изменяем локальную переменную cnt
							// определённую в функции main()
					}
					return a->age < b->age;
				});
	std::cout << oldest->age << std::endl; // 25
	std::cout << cnt; // 2
}
```

Глава 10. Модульный подход

Модульный подход применяется при разработки крупных программ. Суть в следующем:

- Большая программа разбивается на отдельные модули
- Каждый модуль разрабатывается и компилируется отдельно
- Различные модули связываются в единую программу

В предыдущем разделе была проделана большая работа и в результате получилась хорошая заготовка для собственного framework’a — набора полезных библиотек. Прежде чем выделять библиотеку в отдельный модуль, рассмотрим этот процесс на примерах поскромнее.

10.1 Разбиение кода на файлы

Допустим математики написали программу, совершающую сложные геометрические расчёты. В программе создаётся глобальная переменная `const double PI = 3.14`, хранящая значение числа `PI`, а также функции `double getR(double d)`, вычисляющая радиус по диаметру, и `double sphereV(double r)`, вычисляющая объем шара по радиусу.

```
const int PI = 3.14;
double getR(double);
double circleS(double);
int main() {
	double d;
	std::cout << "enter diametr: ";
	std::cin >> d;
	std::cout << getR(d);
}
double getR(double d) {
	return d / 2;
}
double sphereV(double r) {
	return (4.0 / 3.0) * PI * r * r * r;
}
```

Для удобства вынесем весь код, кроме функции `main()` в отдельный файл `mathlib.h`. В итоге получится два файла:

mathlib.h

```
const int PI = 3.14;

double getR(double);
double circleS(double);

double getR(double d) {
	return d / 2;
}

double sphereV(double r) {
	return (4.0 / 3.0) * PI * r * r * r;
}
```

main.cpp

```
#include <iostream>
#include "mathlib.h"

int main() {
	double d;
	std::cout << "enter diametr: ";
	std::cin >> d;
	std::cout << "Radius: " << getR(d);
}
```

Чтобы использовать функции `getR()` и `sphereV()` в своей программе, необходимо подключить заголовочный файл `mathlib.h` через директиву препроцессора `#include`. Обратите внимание, что файл `mathlib.h` во время компиляции должен находиться в том же самой папке, что и `main.cpp`. Название заголовочного файла `"mathlib.h"` записывается в двойных кавычках. Расширение `.h`, подсказывает, что файл является заголовочным `header`.

Заголовочные файлы содержат объявления, (иногда и определения) переменных и функций. Треугольные скобки используются для подключения стандартных библиотек, например, `<iostream>`.

10.2 Компиляция

Компиляция, как упоминалось ранее, состоит из четырех этапов:

1. Препроцессинг
2. Ассемблирования
3. Компиляция кода ассемблера
4. Компоновка

Рассмотрим эти этапы по отдельности.

10.2.1 Препроцессинг

Перед компиляцией исходный код программы обрабатывается текстовым препроцессором, который совершает:

1. Замену комментариев на пустые строки.
2. Текстовые включения директивами `#include`.
3. Выполняет макроподстановки `#define` (на самостоятельное изучение).

Увидев директиву `#include` препроцессор просто добавляет всё содержимое из указанного файла в текущее место. Увидеть результат работы препроцессора можно, указав в команде `g++` флаг `-E`. Указание флага `-E` остановит процесс компиляции после выполения препроцессинга. Результат работы текстового препроцессора запишется в указанный после `-o` файл. Компиляции при этом не произойдёт:

```
g++ -E main.cpp -o main.ii
```

Содержимое получится следующим:

main.ii

```
// много кода из библиотеки <iostream>
// а точнее 32255 строчек кода

# 2 "main.cpp" 2
# 1 "mathlib.h" 1

# 1 "mathlib.h"
const int PI = 3.14;



double getR(double);
double circleS(double);

double getR(double d) {
        return d / 2;
}

double sphereV(double r) {
        return (4.0 / 3.0) * PI * r * r * r;
}
# 3 "main.cpp" 2

int main() {
        double d;
        std::cout << "enter diametr: ";
        std::cin >> d;
        std::cout << "Radius: " << getR(d);
}
```

10.2.2 Ассемблирование

Перевод готового высокоуровневого кода, обработанного препроцессором, в код ассемблера выполняется командой `g++` с флагом `-S`. Указание флага `-S` остановит процесс компиляции после выполения ассемблирования.

```
g++ -S main.ii -o main.s
```

Получится текстовый файл примерно со следующим содержимым:

main.s

```
main:
.LFB1763:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        pushq   %rbx
        subq    $24, %rsp
        .cfi_offset 3, -24
        movq    %fs:40, %rax
        movq    %rax, -24(%rbp)
        xorl    %eax, %eax
        leaq    .LC2(%rip), %rax
        movq    %rax, %rsi
        leaq    _ZSt4cout(%rip), %rax
        movq    %rax, %rdi
        call    _ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc@PLT
        leaq    -32(%rbp), %rax
        movq    %rax, %rsi
        ...
        ...
```

10.2.3 Компиляция кода ассемблера

Указание флага `-c` остановит процесс компиляции после создания объектного файла:

```
g++ -c main.s -o main.o
```

В результате компиляции получим объектный файл `main.o` с объектным кодом следующего содержания:

Объектный код — машинный код с частичным сохранением символьной информации об объявленных переменных, функциях и прочих программных сущностей, необходимой на следующем этапе — компоновке.

10.2.4 Компоновка

Компоновку называют **линковкой** (от англ. linkage). Лучше всего данный этап характеризуется термином **связывание**. Отдельные объектные файлы `.o` связываются в единый исполняемый файл `main.exe`:

```
g++ main.o -o main.exe
```

Все три процесса обычно объединяют в один, который назвали **Компиляция**. Чтобы запустить **Компиляцию**, следует передать компилятору `g++` файл `main.cpp` и не указывать дополнительных флагов, кроме `-o`: `g++ main.cpp -o main.exe`

При выполнении данной команды файлы `main_pp.cpp` и `main.o` не видно, т. к. они являются временными и, возможно, даже не сохраняются на жёсткий диск.

В команде `g++ main.cpp` указан и компилируется только один единственный файл `main.cpp`. Содержимое файла `mathlib.h` уже было включено в файл `main.cpp` на этапе

препроцессинга. Как физические объекты заголовочные `.h` обычно не компилируются, они включаются в файлы `.cpp`.

10.3 Объявление и определение

Допустим физики захотели использовать библиотеку `mathlib` для вычисления массы планеты. Создали функцию `calcPlanetWeight()`, принимающую радиус и среднюю плотность планеты, а возвращающую её массу:

```
#include <iostream>
#include "mathlib.h"
double calcPlanetWeight(double R, double r);
int main() {
	double d = 1'2742'000; // диаметр планеты Земля
	double r = 5520; // средняя плотность планеты Земля
	double R = getR(d); // вычисляем радиус планеты
	std::cout << "Earth Weight: " << calcPlanetWeight(R, r) << std::endl; // 5.70981e+24
}
double calcPlanetWeight(double R, double r) {
	double v = sphereV(R);
	double weight = v * r;
	return weight;
}
```

Научные расчёт выполнились, отлично.

Физик думает: «чем я хуже математиков?». И решает вынести свою программу в отдельный модуль `astronomy.h`:

astronomy.h

```
#include "mathlib.h"

double calcPlanetWeight(double R, double r);

double calcPlanetWeight(double R, double r) {
	double radius = getR(diametr);
	double v = sphereV(radius);
```

```
    double weight = v * r;
    return weight;
}
```

main.cpp

```
#include <iostream>
#include "mathlib.h"
#include "astronomy.h"
int main() {
    double d = 1'2742'000; // диаметр планеты Земля
    double r = 5520; // средняя плотность планеты Земля
    double R = getR(d); // вычисляем радиус планеты
    std::cout << "Earth Weight: " << calcPlanetWeight(R, r) << std::endl; // 5.70981e+24
}
```

Результат компиляции будет включать множество ошибок:

```
In file included from astronomy.h:1,
                 from main.cpp:3:
mathlib.h:1:11: error: redefinition of ‘const int PI’
    1 | const int PI = 3.14;
      |           ^~
In file included from main.cpp:2:
mathlib.h:1:11: note: ‘const int PI’ previously defined here
    1 | const int PI = 3.14;
      |           ^~
In file included from astronomy.h:1,
                 from main.cpp:3:
mathlib.h:6:8: error: redefinition of ‘double getR(double)’
    6 | double getR(double d) {
      |        ^~~~
In file included from main.cpp:2:
mathlib.h:6:8: note: ‘double getR(double)’ previously defined here
    6 | double getR(double d) {
      |        ^~~~
In file included from astronomy.h:1,
```

```
                from main.cpp:3:
mathlib.h:10:8: error: redefinition of ‘double sphereV(double)’
  10 | double sphereV(double r) {
     |        ^~~~~~~
In file included from main.cpp:2:
mathlib.h:10:8: note: ‘double sphereV(double)’ previously defined here
  10 | double sphereV(double r) {
```

Все они выгладят как `error: redefinition of`. Дело в том, что файл `mathlib.h` после замены препроцессором директив `#include`, включился два раза. Первый раз в файле `main.cpp`, второй раз в файле `astronomy.h`. Получилось, что все функции и переменная `const double PI` определены в итоговом файле дважды. Стоит различать понятие объявление и определение:

1. Для функции объявление выглядит следующим образом:

```
double getR(double d);
```

Имена аргументов указывать необязательно:

```
double getR(double);
```

Определение выглядит так:

```
double getR(double d) {
    return d / 2;
}
```

2. С переменной дело обстоит сложнее. Объявление и определение выглядят одинаково:

```
int a; // это и объявление и определение одновременно
```

Существует способ объявить переменную без определения. Используют ключевое слово `extern`:

```
extern int a; // переменная объявлена, но не определена.
```

Подразумевается, что где-то ниже или выше переменная a будет определена:

```
int a;
```

Объявлений у переменной или функции может быть множество. Определение должно быть только одно. Вынесем все определения в отдельные файлы `mathlib.cpp` и `astronomy.cpp`. В заголовочных `.h` файлах оставим только объявления. Получим целых пять файлов:

mathlib.h

```
extern const int PI; // extern означает объявление без определения

double getR(double);
double sphereV(double);
double absolute(double);
```

astronomy.h

```
#include "mathlib.h"

double calcPlanetWeight(double R, double r);
```

mathlib.cpp

```
#include "mathlib.h"

const int PI = 3.14; // определение ранее объявленной переменной

double getR(double d) {
	return d / 2;
}

double sphereV(double r) {
	return (4.0 / 3.0) * PI * r * r * r;
}
double abs(double x) {
	if(x < 0) {
		x *= -1;
	}
	return x;
}
```

astronomy.cpp

```
#include "astronomy.h"

double calcPlanetWeight(double R, double r) {
	double v = sphereV(R);
	double weight = v * r;
	return weight;
}
```

main.cpp

```
#include <iostream>
#include "mathlib.h"
#include "astronomy.h"

int main() {
	double d = 1'2742'000; // диаметр планеты Земля
	double r = 5520; // средняя плотность планеты Земля
	double R = getR(d);
	std::cout << "Earth Weight: " << calcPlanetWeight(R, r) << std::endl;
}
```

В библиотеку `mathlib` была добавлена ещё одна полезная функция `double abs(double)`, которая вычисляет модуль числа, т. е. абсолютное значение.

После выполнения команды:

```
g++ -E main.cpp -o main_pp.cpp
```

Препроцессор выдаст компилятору следующий исходный код:

```
// код библиотеки <iostream>
// #include "mathlib.h"
extern const int PI; // extern означает объявление без определения
double getR(double);
double sphereV(double);
// #include "astronomy.h"
```

```
        // #include "mathlib.h"
extern const int PI; // extern означает объявление без определения
double getR(double); // Повторные объявления
double sphereV(double); // Объявления могут повторяться
double calcPlanetWeight(double R, double r);

int main() {
        double d = 1'2742'000; // диаметр планеты Земля
        double r = 5520; // средняя плотность планеты Земля
        double R = getR(d);
        std::cout << "Earth Weight: " << calcPlanetWeight(R, r) << std::endl;
}
```

Повторные определения отсутствуют, одни объявления.

10.4 Компоновка

При попытке компиляции командой:

```
g++ main.cpp -o main.exe
```

появятся ошибки:

```
/usr/bin/ld: /tmp/ccjvcaHc.o: in function `main':
main.cpp:(.text+0x31): undefined reference to `getR(double)'
/usr/bin/ld: main.cpp:(.text+0x6d): undefined reference to `calcPlanetWeight(double, double)'
collect2: error: ld returned 1 exit status
```

Переводится как, не найдена ссылка на `getR(double)`.

На самом деле компиляция файла `main.cpp` прошла без ошибок. Они возникли после компиляции на этапе компоновки. Файлы программы компилируются отдельно. После компиляции всех файлов, компоновщик находит для каждого объявления, заветное и единственное определение и формирует исполняемый файл `main.exe`.

В нашем случае определения были вынесены в отдельные файлы `mathlib.cpp` и `astronomy.cpp`. Данные файлы не были скомпилированы, следовательно компоновщик не может найти определения для функций и переменных.

Скомпилируем эти файлы по отдельности:

```
g++ main.cpp -o main.o
g++ mathlib.cpp -o mathlib.o
g++ astronomy.cpp -o astronomy.o
```

А затем выполним компоновку:

```
g++ main.o mathlib.o astronomy.o -o main.exe
```

Компоновщик, найдёт в объектных файлах `mathlib.o` и `astronomy.o` требуемые определения, свяжет их с объявлениями, присутствующими в объектном файле `main.o` и сформирует исполняемый файл `main.exe`.

Можно обойтись и одной командой, отправив компилятору сразу все файлы `.cpp`:

```
g++ main.cpp mathlib.cpp astronomy.cpp
```

Итак, компилятор `g++.exe` выполняет три действия:

1. **Препроцессинг** — обработка директив, начинающихся с `#`, например, `#include`, `#define`, `#pragma`.

2. **Ассемблирование** файлов с высокоуровневым исходным кодом, в файлы с ассемблерным кодом `.s` (допустим, что они являются временными, поэтому их не видно).

3. **Компиляция** файлов с ассемблерным кодом в объектные файлы `.o`.

4. **Компоновка** объектных файлов в единый исполняемый модуль `.exe`

В файлы с расширением `.h` помещают только объявления, такие файлы называются **заголовочными файлами**, они описывают, какие сущности имеются в модуле.

Файлы с расширением `.cpp` содержат определения или реализации сущностей, объявленных в заголовочных файлах, и называются **файлами с реализацией**.

Так мы подружили математиков и физиков.

Но в мире есть ещё экономисты.

10.5 Пространства имён

Допустим экономист использует библиотеку `mathlib` в своей повседневной деятельности. Ему оттуда приглянулась функция `abs`. В программе экономист придумал удобную функцию `getR(double)`, которая переводит доллары в рубли и активно её использует:

```
#include <iostream>
#include "mathlib.h"
double course = 80.556; // курс доллара 1 доллар = 80.556 рублей
// переводит доллары в рубли
double getR(double dollar) {
	return course * dollar;
}
int main() {
	double profit_dollars = 12'742'000; // прибыль в долларах
	double profit_rubles = getR(profit_dollars);
}
```

Компилируем программу и модуль `mathlib`:

```
g++ main.cpp mathlib.cpp
```

Получаем ошибку компоновщика:

```
/usr/bin/ld: /tmp/ccBsKw4T.o: in function `getR(double)':
mathlib.cpp:(.text+0x0): multiple definition of `getR(double)'; /tmp/cc7llvir.o:main.cpp:(.text+0x0): first defined here
collect2: error: ld returned 1 exit status
```

Множественное определение `getR(double)`. Экономист придумал функцию с точно такой же сигнатурой как и у функции из библиотеки `mathlib`. Конфликт имён — когда разные сущности, в нашем случае функции, имеют одинаковое имя. Пространство имён — конструкция, которая позволяет избегать конфликта имён.

Синтаксис для создания пространства имён следующий:

```
namespace my_name {
	int a;
}
```

Доступ к переменной `a` теперь осуществляется через пространство имён `my_name` с использованием оператор разрешения области видимости `::`

```
my_name::a = 5;
```

Внутри пространства имён `namespace my_name { }` оператор разрешения области видимости, чтобы обратится к переменной `a` не нужен.

Экономист мог бы заключить свой код в собственное пространство имён, тогда конфликт был бы исчерпан:

```
#include <iostream>
#include "mathlib.h"
namespace eco {
	double course = 80.556; // курс доллара 1 доллар = 80.556 рублей
	// переводит доллары в рубли
	double getR(double dollar) {
		return course * dollar;
	}
}
int main() {
	double profit_dollars = 12'742'000; // прибыль в долларах
	double profit_rubles = eco::getR(profit_dollars);
}
```

Но это не совсем правильно. Кто только не будет ещё пользоваться нашей библиотекой `mathlib`. Лучше её заключить в пространство имён. Главное придумать красивое название из трёх букв, например, `mtb`:

mathlib.h

```
namespace mtb {
	extern const int PI; // extern означает объявление без определения

	// Получение радиуса из диаметра d
	double getR(double d);

	// Вычисление объема сферы по радиусу r
	double sphereV(double r);

	// Находит модуль числа x
	double abs(double x);
}
```

mathlib.cpp

```
#include "mathlib.h"

const int mtb::PI = 3.14;

double mtb::getR(double d) {
	return d / 2;
}

double mtb::sphereV(double r) {
	return (4.0 / 3.0) * PI * r * r * r;
}

double mtb::abs(double x) {
	if(x < 0) {
		x *= -1;
	}

	return x;
}
```

main.cpp

```
#include <iostream>
#include "mathlib.h"

double course = 80.556; // курс доллара 1 доллар = 80.556 рублей

// переводит доллары в рубли
double getR(double dollar) {
	return course * dollar;
}

int main() {
	double profit_dollars = 12'742'000; // прибыль в долларах
	double profit_rubles = getR(profit_dollars); // перевод в рубли
	double module = mtb::abs(-125); // модуль числа
}
```

Теперь все работает. Экономист спокойно переводит доллары в рубли. Если ему вдруг потребуется найти радиус, то он напишет `mtb::getR(double)`.

Конфликт исчерпан.

10.6 Директива using namespace

С первой программы `helloWorld` активно использовалось пространство имён `std`. В учебных целях иногда включают все имена из пространства имён `std::` в глобальную область. Глобальная область — область, которая не включённая ни в одно пространство имён. Для этого используют директиву `using namespace std:`

```
#include <iostream>
using namesppace std;
int main() {
	cout << "Hello, world!";
}
```

Удобно, не нужно писать каждый раз `std::`

Покажем, почему использовать `using namespace` плохо:

```
#include<iostream>
#include <cmath>
#include "mathlib.h"

using namespace std;
using namespace mtb;

int main() {
        abs(-5); // ??? std::abs() или mtb::abs()
}
```

При компиляции возникнет ошибка:

```
main.cpp: In function ‘int main()’:
main.cpp:9:12: error: call of overloaded ‘abs(double)’ is ambiguous
   9 |        abs(-5.6); // ??? std::abs() или mtb::abs()
     |        ~~~^~~~~~
In file included from /usr/include/c++/11/cstdlib:75,
                 from /usr/include/c++/11/ext/string_conversions.h:41,
                 from /usr/include/c++/11/bits/basic_string.h:6608,
                 from /usr/include/c++/11/string:55,
                 from /usr/include/c++/11/bits/locale_classes.h:40,
                 from /usr/include/c++/11/bits/ios_base.h:41,
                 from /usr/include/c++/11/ios:42,
                 from /usr/include/c++/11/ostream:38,
                 from /usr/include/c++/11/iostream:39,
                 from main.cpp:1:
/usr/include/stdlib.h:848:12: note: candidate: ‘int abs(int)’
  848 | extern int abs (int __x) __THROW __attribute__ ((__const__)) __wur;
      |            ^~~
In file included from /usr/include/c++/11/cstdlib:77,
                 from /usr/include/c++/11/ext/string_conversions.h:41,
                 from /usr/include/c++/11/bits/basic_string.h:6608,
                 from /usr/include/c++/11/string:55,
                 from /usr/include/c++/11/bits/locale_classes.h:40,
```

```
                 from /usr/include/c++/11/bits/ios_base.h:41,
                 from /usr/include/c++/11/ios:42,
                 from /usr/include/c++/11/ostream:38,
                 from /usr/include/c++/11/iostream:39,
                 from main.cpp:1:
/usr/include/c++/11/bits/std_abs.h:103:3: note: candidate: ‘constexpr __float128 std::abs(__float128)’
  103 |   abs(__float128 __x)
      |   ^~~
/usr/include/c++/11/bits/std_abs.h:85:3: note: candidate: ‘constexpr __int128 std::abs(__int128)’
   85 |   abs(__GLIBCXX_TYPE_INT_N_0 __x) { return __x >= 0 ? __x : -__x; }
      |   ^~~
/usr/include/c++/11/bits/std_abs.h:79:3: note: candidate: ‘constexpr long double std::abs(long double)’
   79 |   abs(long double __x)
      |   ^~~
/usr/include/c++/11/bits/std_abs.h:75:3: note: candidate: ‘constexpr float std::abs(float)’
   75 |   abs(float __x)
      |   ^~~
/usr/include/c++/11/bits/std_abs.h:71:3: note: candidate: ‘constexpr double std::abs(double)’
   71 |   abs(double __x)
      |   ^~~
/usr/include/c++/11/bits/std_abs.h:61:3: note: candidate: ‘long long int std::abs(long long int)’
   61 |   abs(long long __x) { return __builtin_llabs (__x); }
      |   ^~~
/usr/include/c++/11/bits/std_abs.h:56:3: note: candidate: ‘long int std::abs(long int)’
   56 |   abs(long __i) { return __builtin_labs(__i); }
      |   ^~~
In file included from main.cpp:3:
mathlib.h:11:16: note: candidate: ‘double mtb::abs(double)’
   11 |       double abs(double x);
      |              ^~~
```

Компилятор не знает какую функцию вызвать. В глобальное пространство имён включены две функции с одинаковыми сигнатурами `abs(double)`. Одна функция из стандартной библиотеки `std::abs(double)`, другая `mtb::abs(double)` из `mathlib`.

Пространство имён придумали, чтобы не возникало подобных конфликтов имён. Директива `using namespace` сводит на нет идею пространства имён в принципе.

Кстати, пространства имён могут распространятся на несколько модулей. Функция `std::abs()` объявлена в файле `<cmath>`, но принадлежит тому же пространству имён, что и функция `std::cout`, объявленная в `<iostream>`. Просто помещайте код в каждом заголовочном файле в `namespace my_lib {…}`.

10.7 Low Voltage Library

Превратим динамический массив `DynamicArr` вместе с набором полезных функций в маленький, но полноценный модуль `Low Voltage Library`. Название низкое напряжение, характеризует, то ли простоту его использования, то ли усилия, затраченные на написание программного кода самого модуля. Вынесем код основных функций в файл `lv.h`, определение структуры `DynamicArray` и перегрузку её методов в файл `darr.h`. Разместим весь код в пространстве имён `lv`, дабы избежать конфликта имён между функциями `Low Voltage Library` и другими библиотеками, например, стандартной библиотекой языка `C++ STL`.

10.7.1 Полный листинг

Представим полный листинг программы с разбиением на файлы. Затем подробнее разберём каждый из них:

lv.h

```
#pragma once

namespace lv
{
	// Поиск максимального из двух значения
	template<typename T> T max(T a, T b) {
		if(a < b) {
			return b;
		}
		return a;
	}

	// Обмен переменных местами
	template <typename T> void swap(T& a, T& b) {
		T temp = a;
		a = b;
		b = temp;
	}

	// Поиск максимального элемента между итераторами begin и end
	// некоторой последовательности
	template <typename T>
	T maxElement(T begin, T end) {
		assert(end - begin > 0); // последовательность не пустая
		T max = begin;
		for(T it = begin; it != end; ++it) {
			if(*it > *max) {
				max = it;
			}
		}
		return max;
	}

	// Поиск максимального элемента между итераторами begin и end
```

```
        // некоторой последовательности. Сравнение элементов осуществляется
        // с использованием функции-компоратора
        template <typename T, typename Comparator>
        T maxElement(T begin, T end, Comparator compare) {
                assert(end - begin > 0); // последовательность не пустая
                T max = begin;
                for(T it = begin; it != end; ++it) {
                        if(compare(max, it)) { // if(max < it)
                                max = it;
                        }
                }
                return max;
        }

        template <typename T>
        void sort(T begin, T end) {
                //...
        }
        template <typename T, typename Comparator>
        void sort(T begin, T end, Comparator compare) {
                //...
        }
}
```

darr.h

```
#pragma once

#include <initializer_list>
#include <cassert>
#include "lv.h"
namespace lv
{
        template <typename T>
        struct DynamicArr {
        public:
```

```
// Конструктор.
// Создает массив, размером size
// Если аргумент не указан, будет пустой массив
DynamicArr(int size = 0) {
	assert(size < capacity);
	arr = new T[capacity]; // Выделяем массив в куче
	this->size = size;
}
// Конструктор.
// Создаёт DynamicArr из инициализатора { , , }, подобно массивам
DynamicArr(std::initializer_list<T> init) {
	DynamicArr(int size = 0); // создаем пустой массив

	// заполняем его значениями из списка инициализации { , , }
	for(auto it = init.begin(); it != init.end(); ++it) {
		push_back(*it);
	}
}

// Копирующий конструктор
// Вызывается при инициализации массива существующим массивом
// DynamicArr a2 = a1; // элементы из a1 копируются в a2
DynamicArr(const DynamicArr& rhs) {
	arr = new T[capacity];
	size = rhs.size;
	for(int i = 0; i < size; ++i) {
		arr[i] = rhs[i];
	}
}

// Деструктор. Удаляет массив из кучи
~DynamicArr() { delete[] arr; }
```

```
// добавляет элемент в конец массива
void push_back(T element) {
	assert(size < capacity);
	arr[size] = element;
	++size;
}

// удаляет элемент с конца массива
void pop_back() {
	assert(size > 0);
	--size;
}

int get_size() const {
	return size;
}
// Перегрузка оператора присваивания
// DynamicArr a2 = a1; // элементы из a1 копируются в a2
DynamicArr& operator=(const DynamicArr& rhs) {
	if(this == &rhs) {
		return  *this;
	}
	size = rhs.size;
	for(int i = 0; i < size; ++i) {
		arr[i] = rhs[i];
	}
	return *this;
}

// Перегрузка оператора квадратные скобки []
// Возвращает ссылку на элемент с индексом index
T& operator[](int index) const noexcept {
	assert(index < size);
	return arr[index];
}
```

```
// Итератор (указатель) на начало массива
T* begin() {
	return &arr[0];
}

// Итератор на (точнее, за) конец массива
T* end() {
	return &arr[size];
}

// Поиск среднего арифметического элементов
// Сработает для типов T,
// для которых определы операторы + и / и static_cast<double>()
// Например для числовых типов
double average() {
	int sum = 0;
	for(int i = 0; i < size; ++i) {
		sum += arr[i];
	}
	return static_cast<double>(sum) / size;
}
// Статическая функция. Возвращает новый массив, в котором
// каждый элемент является суммой элементов из массивов a1 и a2
// Для вызова используйте название класса, а не тип
// DynamicArr a3 = DynamicArr::sumArr(a1, a2);
// Размер a3 равен максимальному из a1 и a2
static DynamicArr sumArr(const DynamicArr& a1, const DynamicArr& a2) {
	DynamicArr res(max(a1.size, a2.size));
	for(int i = 0; i < res.size; ++i) {
		res.arr[i] = a1[i] + a2[i];
	}
	return res;
}
```

```
        // Приватные переменные делают класс надёжным
        private:
                int capacity = 1000;
                int size;
                T* arr;

};  // Конец структуры DynamicArr

// Определение перегрузок операторов для DynamicArr и вспомогательных функций     //
Перегрузка оператора+. Делает тоже, что и функция sumArr
template <typename T>
DynamicArr<T> operator+(DynamicArr<T>& left, DynamicArr<T>& right) {
        return DynamicArr<T>::sumArr(left, right);
}

// Перегрузка оператора вывода на экран
template <typename T>
std::ostream& operator<<(std::ostream& os, const DynamicArr<T>& a) {
        bool isFirst = true;
        os << "[";
        for(int i = 0; i < a.GetSize(); ++i) {
                if(isFirst) {
                        isFirst = false;
                }
                else {
                        os << ", ";
                }
                os << a[i];
        }
        os << "]";
        return os;
}
```

```
        // Заполняет все элемнты массива a по порядку, начиная со значения с start
        void fillByOrder(DynamicArr<int>& a, int start = 1) {
                for(int i = 0; i < a.GetSize(); ++i) {
                        a[i] = i + start;
                }
        }
} // закрытие namespace lv {
```

participant.h

```
#pragma once

struct Participant{
        int id;
        int age;
        int grade;
        double totalGrade() const {
                return static_cast<double>(grade) / age;
        }
};

// Перегрузки операторов сравнения. Участники сравниваются по итоговому баллу
bool operator<(const Participant& lhs, const Participant& rhs) {
        return lhs.totalGrade() < rhs.totalGrade();
}

bool operator>(const Participant& lhs, const Participant& rhs) {
        return rhs.totalGrade() < lhs.totalGrade();
}

bool operator==(const Participant& lhs, const Participant& rhs) {
        return (lhs.totalGrade() - rhs.totalGrade()) * (lhs.totalGrade() - rhs.totalGrade()) < 1e-10;
}
```

```
bool operator!=(const Participant& lhs, const Participant& rhs) {
    return !(lhs == rhs);
}

bool operator>=(const Participant& lhs, const Participant& rhs) {
    return !(lhs<rhs);
}

bool operator<=(const Participant& lhs, const Participant& rhs) {
    return !(lhs>=rhs);
}

bool CompareByAge(Participant* lhs, Participant* rhs) {
    return lhs->age < rhs->age;
}
```

main.cpp

```
#include <iostream>
#include "darr.h" // в т.ч. #include "lv.h"
#include "lv.h" // повторное включение не срабатывает из-за #pragma once
#include "participant.h"

int main() {
    // Создание целочисленного массива
    lv::DynamicArr<int> d = {1, 2, 7, 4, 2};
    std::cout << d << std::endl; // [1, 2, 7, 4, 2]

    // поиск максимального элмента
    int* max = lv::maxElement(d.begin(), d.end());
    std::cout << *max; // 7
```

```
    // Создание массива участников
    lv::DynamicArr<Participant> p = {
                                        {1, 15, 100},
                                        {2, 18, 100},
                                        {3, 20, 100},
                                        {4, 19, 100},
                                        {5, 14, 100}
                                    };

    // Поиск победителя (участника с максимальным totalGrade())
    // При сравнении используется перегрузка operator< для Participant

    Participant* win = lv::maxElement(p.begin(), p.end());

    // Поиск самого старшего участника, при сравнении используется
    // компjратор, переданный третьим аргументом в виде лямбда функции
    Participant* oldest =  lv::maxElement(p.begin(), p.end(),
                                    [](auto p1, auto p2) {
                                        return p1->age < p2->age;
                                    });

    std::cout << win->id << std::endl; // 5
    win->grade = 70;
    std::cout << p[4] << std::endl; // 70
    std::cout << oldest->age << std::endl; // 20

}
```

10.7.2 Основные функции

Основные функции библиотеки вынесены в файл `lv.h`. Шаблонная функция `max<>()` позволяет определить максимальное из двух значений. Шаблонная функция `maxElement<>()` находит максимальный элемент последовательности, в т. ч. массива. Обратите внимание, что функция возвращает указатель на найденный элемент. Если изменить значение через указатель (разыменовав его `->`), то будет изменено значение в исходном массиве. В конце функции `main()` через указатель `win` изменяется балл `grade`, набранный победителем (участник с `id = 5`) в массиве `p`.

Шаблонная функция `swap<>()` меняет значения переменных любого типа местами, предварительно скопировав одно из них в третью переменную. Если типы имеют большой размер, то разумнее реализовать менее универсальную, но более эффективную по скорости выполнения функцию. Например, в случае с типом `DynamicArr<>` следует не копировать все элементы `p1` в `temp` и `p2` по одному, а достаточно всего лишь скопировать указатели.

Универсально != эффективно

10.7.3 Инкапсуляция структуры

Структура `DynamicArr` претерпела ряд изменений. Добавились ключевые слова `public:` и `private:` . Они являются модификаторами доступа. **Модификаторы доступа** определяют к каким членам структуры (полям и методам), имеется доступ извне. Поля и методы после `public:` являются открытыми или публичными. Публичные члены структуры можно прочитать или изменить извне структуры. Поля и методы после `private:` являются скрытыми или приватными. Приватные члены доступны только внутри структуры, в которой они объявлены. По умолчанию, все поля структуры публичные:

```
struct Counter {
	int cnt;
};
тоже самое:
struct Counter {
public:
	int cnt;
};
```

Объявим переменную счётчик типа `Counter`, по умолчанию в `cnt` запишется `0`.

```
Counter c;
```

Следующее выражение «ломает» счётчик, т. к. подразумевается, что счётчик может только увеличиваться на единицу.

```
c.cnt += 5; // ok, but bad
```

Укажем перед объявлением переменной `cnt` модификатор доступа `private`:

```
struct Counter {
private:
    int cnt;
};
Counter c;
c.cnt = 5; // error
```

Компилятор выдаст следующую ошибку:

```
main.cpp: In function ‘int main()’:
main.cpp:209:11: error: ‘int Counter::cnt’ is private within this context
 209 |       c.cnt;
```

Как теперь изменять счётчик? Добавим в структуру `Counter` публичный метод `count()`, который будет увеличивать приватное поле `cnt` на единицу:

```
struct Counter {
    public:
        void count() {
            ++cnt;
        }
    private:
        int cnt;
    };
    Counter c;
    c.count(); // ok
```

Добавим метод `view()`, который позволит узнать текущее значение счётчика:

```
class Counter {
public:
    void count() {
          ++cnt;
    }
    int view() {
          return cnt;
    }
private:
    int cnt;
};
Counter c;
c.count();
c.count();
std::cout << c.view(); // 2
```

Вместо `struct` можно указать ключевое слово `class`. Между `class` и `struct` в языке `C++` существует единственное отличие: у классов поля по умолчанию приватные, а у структур открытые. В других языках, например, `C#`, разница между `class` и `struct` более значительная.

Ключевое слово `struct` при объявлении используют, если описываемый тип представляет собой данные, состоящие из нескольких значений, например, структура точка:

```
struct Point { int x, y };
```

Более сложные типы принято помечать ключевым словом `class`. Принципиальная разница между `struct` и `class` отсутствует, кроме представленной выше.

Таким образом, `DynamicArr<>` превратился в класс с приватными полями `size`, `arr`, `capacity`. Все методы класса были сделаны публичными. Сокрытие полей позволяет избегать случаев некорректного использования класса:

```
DynamicArr<int> a = {1, 2, 3, 4, 5};
a.size = a.capacity + 1; // размер массива size становится больше его вместимости capacity
a.arr = new int[a.size];// утечка памяти
                        // в arr записывается новый массив, старый при этом удалить забыли
```

Данный код приведёт к ошибкам, т. к. `size` и `arr` объявлены с модификатором `private:`.

Переменная `size` устанавливается конструктором, и далее может увеличиваться при вызове метода `push_back()` или уменьшаться при вызове метода `pop_back()`. Новый метод `get_size()` позволяет узнать размер массива:

```
for(int i = 0; i < a.get_size(); ++i) { ... }
```

Инкапсулирование — сокрытие реализации класса от внешнего мира. Приведём хорошее правило для разработки классов:

Все поля всегда делайте приватными. Методы тоже делайте приватными. Публичными оставляете только те методы, которые необходимы для взаимодействия с классом извне.

Как же теперь обратиться к элементу массива:

```
a.arr[4] = 5; // error, т. к. arr приватный
```

Один из способов добавить публичный метод, который будет получать индекс, а возвращать ссылку на элемент (именно ссылку, а не значение):

```
class DynamicArr {
public:
    T& at(int index) {
        return arr[index];
    }
}
a.at(4) = 7;    // сработает, at() возвращает ссылку на ячейку памяти,
                // в которой хранится arr[index]
```

В нашем классе мы поступили более красиво.

10.7.4 Перегрузка operator[]

Оказывается можно перегружать не только арифметические операторы и операторы сравнения, но и все остальные. Перегрузим оператор прямоугольные скобки `operator[]`, чтобы внешне работа с переменной типа `DynamicArr<>` выглядела, как работа с обыкновенным массивом:

```
// Перегрузка оператора квадратные скобки []
// Возвращает ссылку на элемент с индексом index
T& operator[](int index) const noexcept {
	assert(index < size && index >= 0);
	return arr[index];
}
a.arr[4] = 5; // ok, функция operator[] возвращает ссылку на массив
```

Обратите внимание, что `assert(index < size && index >= 0);` предостерегает от выхода за границы массива. Полезная проверка, которая отсутствует для обыкновенных массивов.

10.7.5 Инициализирующий конструктор

В класс `DynamicArr` был добавлен инициализирующий конструктор:

```
DynamicArr(std::initializer_list<T> init) {
	DynamicArr(int size = 0); // создаем пустой массив
	// заполняем его значениями из списка инициализации { , , }
	for(auto it = init.begin(); it != init.end(); ++it) {
		push_back(*it);
	}
}
```

Данный конструктор вызывается, при создании объекта через инициализатор `{, ,}`:

```
DynamicArr<int> a = {1, 2, 3, 4};
```

Значения перечисленные в фигурных скобках передаются в переменную `init` специального типа `std::initializer_list<T>`. Объекты (т. е. переменные) данного типа имеют два итератора `begin()` и `end()`, позволяющие перебрать и скопировать все значения из инициализатора в массив `arr`. Для этого можно написать цикл:

```
for(auto it = init.begin(); it != init.end(); ++it) {
    arr[size] = *it;
    ++size;
}
```

но логичнее воспользоваться уже готовым методом `push_back()`, что и было сделано.

Аналогично, вместо двух строчек:

```
arr = new T[capacity]; // Выделяем массив в куче
this->size = 0;
```

лаконичнее вызвать готовый простой конструктор:

```
DynamicArr(int size = 0); // создаём пустой массив
```

10.7.6 Структура Participant

Описание структуры `Participant` было вынесено в отдельный файл. Туда же были отправлены и все перегруженные операторы сравнения. Ключевое слово `struct` было оставлено, т. к. предназначение `Participant` — контейнер для хранения данных об одном участнике. Метод `totalGrade()` настолько простой, что ключевое слово `struct` на `class` можно не менять. Структура `Participant` не какой то сложный тип со своей логикой, как у `DynamicArr`.

10.7.7 Директива #pragma once

Ранее был рассмотрен способ избежать повторных включений. Для этого в файле `.h` оставляли только объявления, а определения переносили в файлы `.cpp`. Существует более простой способ избежать повторного включения. Достаточно написать в начале файла директиву `#pragma once`. Препроцессор позаботится, чтобы повторные `#include` не испортили работу компилятора.

С т.з. логики программы, разделение объявления и определения в отдельные файлы имеет практическую пользу. Например, можно разбить большую программу на отдельные модули. Разрабатывать и компилировать `.cpp` файлы модулей отдельно друг от друга. Более того, для

одного заголовочного файла с объявлениями `.h` можно реализовать несколько параллельных файлов `.cpp` с реализациями. При компиляции выбирать, какую из реализаций использовать.

Для наших скромных целей `#pragma once` достаточно.

10.8 Обзор

Примерный план дальнейшего изучения может быть следующим:

1. Низкоуровневое программирование
2. Объектно-ориентированное программирование
3. Структуры данных
4. Стандартная библиотека `STL`, контейнеры и алгоритмы
5. …

10.8.1 Низкоуровневое программирование

Для начала неплохо было бы познакомится с низкоуровневым программированием. Потрогать, в прямом смысле этого слова, железо. Написать несколько программ на ассемблере. На первом этапе, полезнее будет написать программу на ассемблере не для большого компьютера, а для микроконтроллера, например `Atmega` из серии `AVR`.

> Существует прекрасная книга «Практическое программирование микроконтроллеров Atmel AVR на языке ассемблера», Ю. Ревич.
>
> Пошаговая инструкция представлена в последнем разделе книги «FreeASM», С. Игонин.
>
> Далее можно ознакомиться с классикой «Архитектура компьютера», Э. Таненбаум, Т. Остин

10.8.2 ООП

Заложив мощный фундамент, можно подниматься выше. Объектно-ориентированное программирование — парадигма программирования, в которой программные сущности выступают в виде объектов. Работа с объектами программы максимально приближена к работе с объектами реального мира. Складывать помидоры с яблоками и получать персики, в ООП это нормально.

Основное понятие объектно-ориентированного программирования **объект**. Сперва описывается класс (т. е. тип), затем создаётся объект типа, описанного ранее.

Объект обладает **свойствами** (т. е. полями) и **действиями** (т. е. методами).

Состояние объекта — совокупность его свойств. Основная идея заключается в следующем, объект может производить действия (вызываются методы), которые изменяют свойства (поля), т. е. состояние объекта. Конкретный объект также называют **экземпляр класса**.

Например, создаётся игра «бродилка». Разрабатывается класс `Hero`. Далее создаётся объект (экземпляр) данного класса `Hero hero`. Герой подбирает монетку номиналом пять рублей `hero.pickUp(5)`, при этом увеличивается количество монеток в его кошельке `hero.cash`. Таким образом, изменяется состояние героя `hero`. Разумеется поле `cash` следует сделать приватным и добавить метод `hero.getCash()`, который позволит узнать, но не изменить количество монеток.

Основные принципы объектно-ориентированного программирования:

1. **Инкапсуляция** — сокрытие всех полей и максимальное сокрытие методов. Инкапсуляция была применена при разработке `DynamicArr`.
2. **Наследование** — создание классов, на основе уже имеющегося. Производный класс при этом, расширяет функциональность базового класса.
3. **Полиморфизм** — объект может выступать сразу в качестве множества типов. Полиморфизм расшифровывается как «много форм».
4. **Абстрагирование** — разработка архитектуры программы с использованием абстрактных классов. Абстрактные классы содержат методы без реализации — абстрактные методы. От абстрактного класса наследуются конкретные классы, которые реализуют абстрактные методы базового класса. Таким образом, можно реализовать интерфейс приложения. **Интерфейс** — внешняя часть, фасад. Например, руль и педали — интерфейс транспортного средства. Интерфейса достаточно, чтобы управлять транспортным средством. А вот в качестве реализации, можно подставить мотоцикл, автомобиль, подводную лодку или самолёт. Интерфейс останется тот же, но действия при нажатии педалей и вращении руля разные.

ООП было придумано для удобства разработки архитектур больших приложений, которыми и являются большинство современных программ.

> Серьёзной книгой по ООП является: «Принципы, паттерны и методики гибкой разработки на языке C#» Р. С. Мартин, М. Мартин.

10.8.3 Структуры данных

Перед использованием контейнеров, следует изучить теорию по структурам данных. Структура данных описывает то, как данные хранятся в памяти компьютера. Самые распространённые структуры данных:

- Массив
- Односвязный список
- Двусвязный список
- Хэш-таблица.

10.8.4 Библиотека STL

Параллельно с ООП можно начать знакомство с библиотекой `STL`, которая в настоящее время считается стандартной библиотекой языка `C++`. С самой первой программы мы активно пользовались данной библиотекой, когда писали `std::что-нибудь`. Библиотека `STL` содержит множество функций и классов. Полезно начать с изучения контейнеров и алгоритмов.

Контейнер — специальный класс для хранения данных, который удобно использовать вместо массива.

Основные контейнеры библиотеки `STL`:

- `vector<T>` — динамический массив, похож на `DynamicArr`.
- `set<T>` — множество — хранит уникальные элементы. Каждое значение встречается только один раз.
- `map<TKey, TValue>` — словарь хранит данные в виде набора пар ключ-значение.
- и множество других

Контейнер `vector<>` реализует хранение данных в виде структуры — массив. Внешне выглядит как динамический массив, но внутри содержит массив бо́льшей ёмкости `v.capacity()`, чем текущее количество элементов `v.size()`.

При попытке добавления элемента в уже заполненный вектор, создаётся массив в два раза бо́льшей ёмкости. Элементы из старого массива копируются в новый массив, память,

выделенная под старый массив освобождается. Примерно это происходит, при вызове `v.push_back(…)`, если размер массива сравнялся с его реальной ёмкостью.

Не следует путать структуры данных и контейнеры. Например, в языке C# имеется контейнер `List<T>`. Не смотря на название, он представлен структурой данных массив. `List<T>` аналог `vector<T>` или нашего `DynamicArr<T>`. В C# для хранения данных в виде реального связного списка имеется контейнер `LinkedList<T>`. В C++ путаница отсутствует, контейнер `list<T>` реализован в памяти в виде односвязного списка.

Библиотека STL содержит множество готовых полезных алгоритмов в модуле `<algorithm>`, все они шаблонные или обобщённые. Алгоритмы работают с последовательностями любого типа. Последовательностью считается всё, для чего имеется итератор, которые желательно тоже изучить. Иными словами, если по последовательности можно пройтись итератором, то применимы множество алгоритмов:

`std::sort(begin, end)` — сортировка последовательности

`std::reverse(begin, end)` — реверс или переворот последовательности

`std::reverse(std::sort(begin, end))` — не самая эффективная сортировка по убыванию

`std::accumulate(begin, end, 0)` — в простейшем случае сумма всех элементов, но может делать и куда более интересные вещи

`std::max_element(begin, end)`

`std::min_element(begin, end)`

`std::swap(a, b)`

Многие функции из `<algorithm>` последним аргументом принимают компаратор или предикат — функции, которые используются при выполнении алгоритма. Здесь бывает удобно использовать **лямбда-функции**.

Полезно будет познакомиться с новой формой цикла `for()`, т. н. **range-based for loop**. С помощью такого цикла вывод всех элементов массива запишется следующим образом:

```
std::vector<int> v = {1, 2, 3, 4, 5};
for(int item : v) {
    std::cout << item;
}
```

Внутри `for(:)` широкое распространение получило применение ключевого слова `auto`, которое по сути является синтаксическим сокращением, а не каким-то особым типом. Компилятор в данном случае определяет тип самостоятельно:

```
for(auto item : v) {
    // ...
}
```

Классикой по C++ является объёмная книга «Язык программирования C++», Б. Страуструп

Дальше, сориентируетесь...

Эпилог

Электронные вычислительные машины были созданы с целью выполнять тяжёлый вычислительный труд, который не под силу человеку.

Например, как человеку невозможно поднять тяжёлую бетонную плиту без помощи крана, так ему невозможно решить сложное уравнение без использования компьютера.

Вычислительные машины помогают людям во многих областях: моделирование различных процессов и устройств, обработка экспериментальных данных, прогнозирование экономических процессов и метеорологических явлений, коммуникация и логистика в процессе производства. Безусловно, информационные технологии определяют промышленный прогресс на текущем этапе развития цивилизации.

Последнее время вычислительные мощности начинают использовать не по назначению. Существуют сферы жизни, в которых цифровые технологии не являются столь необходимыми, а иногда даже приносят очевидный вред.

Если вы избрали путь программиста, хочется верить, что он будет приносить благо этому миру. Для этого следует ознакомиться с проблемой досконально, начиная с железа, заканчивая «сверхвысокоуровневыми» языками. Мало быть просто программистом. Программирование ради программирования бессмысленно. Следует стремиться стать профессионалом в той предметной области, в которой вы окажетесь: физика, медицина, экономика, оборонная промышленность, коммуникации, архитектура и т.д.

Главное оставаться Человеком, а вычислительные мощности использовать по назначению.

Препятствия преодолимы, верьте в себя!