Содержание

1. Первая программа на C++

1.1. Минимальная программа

Пример минимальной программы:

```
1  int main()
2  {
3      // nothing interesting in this program...
4
5      /*
6          Seriously, we doesn't have even message about Hello
               World!
7          I hope that we fix this in another section...
8      */
9      return 0;
10 }
```

В данном примере представлена функций `main`, которая является точкой входа программы. Может быть сколько угодно функций в программе, но функция `main` обязана присутствовать среди них, причем в единственном экземпляре! Исполнение всей программы начинается именно с этой функции. `int` перед названием функции говорит о том, что данная функция должна вернуть целочисленное значение. `main` возвращает результат выполнения программы — `return 0` означает, что программа выполнилась без ошибки и корректно завершила свою работу. В противном случае необходимо вернуть ненулевое значение.

Обратите внимание, что при помощи // можно комментировать программу — то есть на данной строке ничего выполняться не будет. Комментарий можно растянуть на несколько линий, обрамив его между /* и */. Примеры комментариев представлены в коде сверху.

1.2. Программа «Hello, world!», запись в консоль данных

Усложним программу, добавив вывод строки «Hello, world!» на экран консольного приложения:

```
1  #include <iostream>
2
3  int main()
4  {
```

```
5       std::cout << "Hello, world!" << std::endl;
6       return 0;
7   }
```

В данной программе, по отношению к предыдущей, добавились две строчки:

- `#include <iostream>`
- `std::cout << "Hello, world!"<< std::endl;`

`#include <iostream>` является директивой (командой) препроцессора. Данные строки обрабатываются им *до* компиляции программы. Данная команда позволяет препроцессору включить в программу содержимое заголовочного файла `iostream`, который отвечает за работу потоков ввода\вывода. Данный файл необходимо включать в любую программу, где ожидается работа с консолью — когда данные считываются с консоли и когда необходимо передать какие-нибудь данные консоли.

Вся строка `std::cout << "Hello, world!"<< std::endl;` является оператором. Каждый оператор в языке оканчивается символом ;.

Ввод\вывод в C++ осуществляется с помощью символьных потоков:

- `std::cin` — стандартный входной поток;
- `std::cout` — стандартный выходной поток;
- `std::cerr` — стандартный поток ошибок.

Обратите внимание, что перед `cin`, `cout` и `cerr` помещено `std::`. Это означает, что данные потоки принадлежат *пространству имен* (*namespace*) `std`. Пространства имен нужны для удобной организации объектов и операторов по смыслу.

Операция `<<` называется *операцией передачи в поток*. При выполнении этой операции то, что стоит справа, передается в поток, который стоит слева. Обратите внимание на направленность данной операции — данные передаются в поток справа налево! `std::endl` является функцией, которая возвращает символ перевода на новую строчку.

Для того, чтобы постоянно не использовать в коде определенное пространство имен, можно воспользоваться двумя методами:

1. Использовать для текущей файла всё пространство имен целиком:

```
1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     cout << "Hello, world!" << endl;
7     return 0;
8 }
```

2. Вынести отдельные объекты и операции:

```
1 #include <iostream>
2 using std::cout;
3 using std::endl;
4
5 int main()
6 {
7     cout << "Hello, world!" << endl;
8     return 0;
9 }
```

Рекомендуется использовать второй метод, если количество объявлений не очень большое, в противном случае не возбраняется подключать все пространство имен.

Записывать в стандартный выходной поток можно не только строки, но и данные других типов:

```
1 #include <iostream>
2 int main()
3 {
4     int a = 5;
5     int b = 7;
6     std::cout << a + b << std::endl;
7 }
```

1.3. Чтение данных с консоли

С помощью потока `std::cin` можно считывать данные, которые вводятся с консоли:

```
1 #include <iostream>
2 int main()
```

```
3  {
4      int a = 0;
5      int b = 0;
6      int c = 0;
7
8      std::cout << "Enter a:" << std::endl;
9      std::cin >> a;
10
11     std::cout << "Enter b:" << std::endl;
12     std::cin >> b;
13
14     std::cout << "a + b = " << a + b << std::endl;
15 }
```

Операция << называется *операцией извлечения из потока*. При выполнении данной операции данные из стандартного входного потока слева будут переданы переменной, которая находится справа.

2. Справочный материал по C++

2.1. Переменные (объявление, типы, операции с переменными)

2.1.1 Объявление переменных

Синтаксис объявления переменных:

`<Modification> <Type> variableName = variableValue;`

`Type` — тип переменной, `Modification` — изменение (модификация), накладываемое на переменную, `variableName` — имя переменной, `variableValue` — значение переменной.

После объявления переменной имеет смысл назначить ей какое-либо допустимое значение. Даже если переменная будет изменяться в дальнейшем коде программы, правилом хорошего тона является выставления значения по умолчанию, в противном случае при выполнении программы в этих переменных может оказаться «мусор» (случайные значения).

Инициализацию переменной можно двумя способами:

- Через оператор `=` :

```
int a = 5;
```

- Через оператор `()` — инициализация переменной с помощью конструктора:

```
int a(5);
```

На одной строке можно объявлять несколько переменных одного типа:

```
1 int p = 2,k = 3;
2 double s,m; // bad thing, because there is no initialization!
```

2.1.2 Типы переменных

Стандартные типы:

1. `bool` — логический тип;
2. `char` — целочисленный тип, используется для представления символов ('a', 'z', '.', ...);

3. `int` — целочисленный тип, используется для хранения **целых** чисел;

4. `float` — вещественный тип, используется для хранения **дробных** чисел;

5. `double` — вещественный тип, используется для хранения **дробных** чисел, обладает большей точностью, чем тип `float`.

Возможные модификаторы:

1. `const` — запрет менять значение переменной в ходе программы. Если попытаться поменять где-нибудь константу, то компилятор сообщит о ошибке:

```
1 const double pi = 3.1415;
2 pi = 3.0; // This is wrong!!!
```

2. `unsigned` — переменная будет принимать только положительные значения;

3. `short`— укорачивает диапазон переменных;

4. `long` — увеличивает диапазон переменных.

Размерности переменных:

Тип	*Размер в байтах*	*Диапазон*
`bool`	1	true/false
`char`	1	$-128 \ldots 127$
`unsigned char`	1	$0 \ldots 255$
`short int`	2	$-32\,768 \ldots 32\,767$
`unsigned short int`	2	$0 \ldots 65\,535$
`int`	4	$-2\,147\,483\,648 \ldots 2\,147\,483\,647$
`unsigned int`	4	$0 \ldots 4\,294\,967\,295$
`long int`	4	$-2\,147\,483\,648 \ldots 2\,147\,483\,647$
`unsigned long int`	4	$0 \ldots 4\,294\,967\,295$
`long long`	8	$-9\,223\,372\,036\,854\,775\,808 \ldots$ $\ldots 9\,223\,372\,036\,854\,775\,807$
`unsigned long long`	8	$0 \ldots 18\,446\,744\,073\,709\,551\,615$
`float`	4	$\pm 3.4 \cdot 10^{\pm 38}$ (7 цифр)
`double`	8	$\pm 1.7 \cdot 10^{\pm 308}$ (15 цифр)

2.1.3 Операции с переменными

Стандартные арифметические операции над переменными:

1. Операция присваивания (=). Значение выражения присваивается переменной:

```
1 int a = 5;
2 int b = 42; // a = 5, b = 42
3 a = b; // a = 42, b = 42
```

2. Операция сложения (+), вычитания (-), умножения (*), целочисленное деление (/), взятие остатка по модулю (%):

```
1 int a = 5;
2 int b = 3;
3
4 int l = 5 + 3 - 1; // l = 7
5 int c = a + b; // c = 8
6 int d = a - b; // d = 2
7 int k = a * b; // k = 15
8 int div = a / b; // div = 1
9 int mod = a % b; // mod = 2
```

   Первоначально обрабатываются операции умножения, деления и взятия остатка по модулю. Если операций несколько, то выполняются слева направо. После этих операций по такому же принципу обрабатываются и операции сложения и вычитания.

3. Операция (). Выражения в скобках оцениваются в первую очередь. В случае вложенных скобок вычисляется значение во внутренних скобках. Если есть одинаковые скобки «одного уровня» — они вычисляются слева направо.

Существуют короткие формы записей арифметических операций:

```
1 a += b --> a = a + b
2 a -= b --> a = a - b
3 a *= b --> a = a * b
4 a /= b --> a = a / b
5 a %= b --> a = a % b
```

В C++ существуют операции инкремента (++) и декремента (--). Данные операции увеличивают или уменьшают значение переменной на 1. Различают пре-

фиксную форму (`++a\--b`) и суффиксную (постфиксную) форму (`a++\b--`). Различие между формами в том, что при использовании префиксной формы значение переменной изменяется сразу же, как программа выполняет данную операцию; в суффиксной — программа сперва использует старое значение, а потом изменяет его.

Операции сравнения над переменными:

1. Операция равенства (`==`). Принимает значение **true**, если оба выражения равны, в противном случае — **false**. *Не следует путать с присваиванием значения переменной!!!*

```
1 bool result = 4 % 2 == 0; // true
2
3 int a = 5;
4 int b = 2;
5 bool newResult = a == b; // false
```

2. Операция неравенства (`!=`). Принимает значение **true**, если оба выражения различны, в противном случае — **false**;

3. Операции отношения (`>`, `<`, `>=`, `<=`). Возвращают значение типа *bool* в зависимости от своей функции:

```
1 bool a = 5 > 3; // true
2 bool b = 4 < 2; // false
3 bool c = 6 <= 6; // true
4 bool d = 5 >= 2; // true
```

Логические операции:

1. Логическое отрицание (`!`) — Инвертирует логическое значение;

2. Логическое умножение, И (`&&`). Если первое выражение равно *false*, то тогда вычисление второго выражения не производится;

3. Логическое сложение, ИЛИ (`||`). Первое выражение вычисляется всегда, второе вычисляется в том случае, если первое выражение принимает значение *fasle*.

Прочие операции:

- `?:` — тернарный оператор, подробнее в 2.2.2;
- `[]` — взятие элемента массива по индексу, подробнее в 2.4.

2.1.4 Приоритеты и ассоциативность операций

Операции	*Ассоциативность*	*Тип*
`() []`	слева направо	наивысший приоритет
`++ --`	слева направо	унарные (постфиксные)
`++ -- !`	справа налево	унарные (префиксные)
`* / %`	слева направо	мультипликативные
`+ -`	слева направо	аддитивные
`<< >>`	слева направо	передача\извлечение из потока
`< <= > >=`	слева направо	отношения
`== !=`	слева направо	равенства
`&&`	слева направо	логическое И
`||`	слева направо	логическое ИЛИ
`?:`	справа налево	условная
`= += -= *= /= %=`	справа налево	присваивания
`,`	слева направо	запятая

2.2. Условный оператор `if`, тернарный оператор, оператор `switch`

2.2.1 Условный оператор `if`

Общая структура условного оператора:

```
1 if (expression_1)
2     statement_1
3 [else
4     statement_2]
```

`expression` — логическое выражение; `statement\_X` — выполняемая операция.

Начало условия задается ключевым словом `if`, после которого в круглых скобках идет логическое выражение типа `bool`. Наличие круглых скобок вне выраже-

ния обязательно, в противном случае компилятор выкинет ошибку. После выражения идут операции, которые следует выполнять в том случае, если выражение в круглых скобках истинно. Если операций несколько, то необходимо поставить фигурные скобки, в противном случае ими можно обойтись.

Пример условия:

```
1  int a = 13;
2  if (a < 16)
3      a = a + 2;
4
5  // a = 15
6
7  int count = 2;
8  if (count > 3)
9  {
10     count++;
11     age += 5;
12 }
13
14 // count = 2, age = 15
```

Если необходимо выполнить набор действий в том случае, когда выражение ложно, то необходимо поставить ключевое слово `else` и перечислить набор операций (если операция одна, то фигурные скобки можно опустить, в противном случае они обязательны). Если условие не предполагает действий в том случае, если выражение в скобках ложно, то нет необходимости ставить `else`.

Пример:

```
1  int a = 5;
2  int b = 0;
3
4  if (a % 2 == 0)
5      b++;
6  else
7  {
8      a--;
9      b--;
10 }
11
12 // b = -1
```

Допускается использования нескольких блоков `if` . . . `else`:

```
int age = 15;

if (age <= 3)
    ...
else if (age > 3 && age <= 7)
    ...
else if (age > 7 && age <= 18)
    ...
else
    ...
```

2.2.2 Тернарный оператор

Тернарный оператор возвращает одно из двух выражений в зависимости от результата логического выражения.

Структура тернарного оператора:

```
expression ? statement_1 : statement_2;
```

Оператор сначала вычисляет выражение `expression`. Если это выражение принимает значение `true`, то результатом выполнения всего оператора будет `statement_1`, а в противном случае — `statement_2`.

Пример:

```
int a = 3, b = 5;
int min = a < b ? a : b;
// a = 3
```

2.2.3 Оператор `switch`

Структура оператора `switch`:

```
switch ( expression )
{
    case constant-expression : statement
    [default    : statement]
}
```

`expression` — выражение, которое можно привести к целочисленному типу; `constant-expression` — константное значение, если выражение будет иметь данное значение, то произойдет обработка блока `statement`. Если ни один из `case`'ов не

сработал, то вычисляется `statement` в блоке `default`, однако данный блок не обязателен к указанию и может быть опущен — в таком случаю программа перейдет к выполнению следующих инструкций после оператора `switch`.

Пример использования:

```
1  int month = 4;
2  switch (month)
3  {
4      case 1:
5          ...
6          break;
7      case 2:
8          ...
9          break;
10     ...
11     case 4:
12         ...
13         break;
14     ...
15     case 12:
16         ...
17         break;
18     default:
19         ...
20 }
```

В данном примере будут выполняться инструкции, которые идут в блоке `case 4:`. Обратите внимание, что после окончания инструкций в блоке стоит оператор `break` — он необходим для прекращения работы оператора `switch` после выполнения команд в блоке, в противном случае оператор `switch` будет продолжать свою работу!

Существует возможность указания нескольких `case`'ов для одного и того же блока команд:

```
1  char answer = 'y';
2  switch (answer)
3  {
4      case 'y':
5      case 'Y':
6          ...
7          break;
8      case 'n':
```

```
9       case 'N':
10          ...
11          break;
12  }
```

2.3. Циклы while, do-while, for

2.3.1 Цикл с предусловием (while)

Структура цикла while:

```
1  while (expression)
2      statement
```

expression — выражение типа bool, задает условие выхода из цикла; statement — набор операций, которые будут выполняться в цикле (если операций больше одной, то нужно заключить их в фигурные скобки, в противном случае делать данное действие необязательно). Цикл while будет выполняться до тех пор, пока expression будет равно true. Программа может не войти в тело цикла (то есть выполнить операции в нем), если изначально expression будет равен false.

Пример:

```
1  int a = 0;
2  int b = 0;
3  while (a < 10)
4  {
5      b *= a*a;
6      a++;
7  }
```

Цикл while можно досрочно завершить при помощи оператора break (управление программы перейдет к следующей инструкции):

```
1
2  char answer = 'v';
3
4  while (a != 'y')
5  {
6      ...
7
8      if (answer == 'q')
9          break;
10
```

```
11      ...
12  }
```

В данном примере выход из цикла произойдет в двух случаях: пока переменная *answer* не примет значение *y*, либо значение *q*. Во втором случае все операции после данного условия не будут выполнены и произойдет выход из цикла.

Если использовать оператор `continue` в цикле, то все операции выполняться также не будут, однако цикл не будет завершен и его выполнение будет продолжено с самой первой операции в цикле.

2.3.2 Цикл с постусловием (`do-while`)

Структура цикла `do-while`:

```
1  do
2      statement
3  while (expression);
```

`expression` — выражение типа `bool`, задает условие выхода из цикла; `statement` — набор операций, которые будут выполняться в цикле (если операций больше одной, то нужно заключить их в фигурные скобки, в противном случае делать данное действие необязательно). Цикл `do-while` будет выполняться до тех пор, пока выражение `expression` будет равно `true`. В данном цикле блок `statement` будет выполняться в первую очередь, условие выхода из цикла будет проверяться после выполнения операций. Операторы `break` и `continue` имеют такой же эффект, что и в цикле `while`.

Пример:

```
1  int a = 0;
2  do
3  {
4      a++;
5  }
6  while (a < 10);
```

2.3.3 Цикл со счетчиком (`for`)

Конструкция цикла `for`:

```
1  for (init-expression ; cond-expression ; loop-expression)
2      statement
```

Обозначения:

- `init-expression` — набор операций, которые выполняются один раз **до** начала цикла, чаще всего используется для инициализации переменных, которые будут использоваться в цикле для подсчета итераций;
- `cond-expression` — выражение типа `bool`, которое задает условие выхода из цикла — проверяется перед каждой итерацией цикла (в том числе и перед самой первой), если *cond-expression* равное `false`, то цикл завершается;
- `loop-expression` — набор операций, который выполняется после итерации цикла, обычно используется для изменения переменных, который подсчитывают количество итераций;
- `statement` — набор операций, которые будут производится в каждой итерации цикла (если операций больше одной, то нужно заключить их в фигурные скобки, в противном случае делать данное действие необязательно).

Пример:

```
1 int sum = 0;
2 const int lim = 10;
3
4 for (int i = 0; i < lim; ++i)
5     sum += i;
```

В блоках `init-expression` и `loop-expression` может быть несколько выражений, разделенных запятой:

```
1 int sum = 0;
2
3 for (int i = 0, j = 0; i + j < lim; ++i, ++j)
4     sum += i + j;
```

В блоке `loop-expression` переменные, ответственные за количество итераций, могут изменяться по-разному:

```
1 for (int i = 10; i > 0; --i)
2 {
3     ...
4 }
5
6 for (int k = 0; k > 200; k += 2)
```

```
7 {
8       ...
9 }
```

Операторы `break` и `continue` имеют такой же эффект, что и в циклах `while` и `do-while`.

2.4. Массивы

2.4.1 Одномерные массивы

Общая структура:

`Type arrayName[arraySize];`

`Type` — тип элементов массива (массив может состоять из целых чисел, дробных, символов, ...); `arrayName` — название массива; `arraySize` — размер массива. Индексы, по которым можно обращаться к элементам массива, лежат в диапазоне $[0 \ldots \text{arraySize} - 1]$.

Обращение к элементу одномерного массива по индексу — `arrayName[index]`.

Инициализация одномерного массива:

- Использовать инициализатор {} при объявлении массива:

  ```
  int costs[4] = {100, 200, 300, 400};
  ```

  В таком случае можно не указывать размер массива:

  ```
  int costs[] = {2, 4, 6};
  ```

- Инициализировать массив при помощи циклов:

  ```
  1 const int arraySize = 5;
  2 int myArray[arraySize];
  3
  4 for (int i = 0; i < arraySize; ++i)
  5     myArray[i] = ...;
  ```

Пример инициализированного массива:

Индекс массива	0	1	2	3
Значение массива	34	-4	11	64

Рекомендуется при работе с одномерными массивами инициализировать их, т.к. в противном случае в элементах массивов будет записан «мусор»!

2.4.2 Двумерные массивы

Общая структура:

`Type arrayName[arraySizeRow][arraySizeColumn];`

`Type` — тип массива (массив может состоять из целых чисел, дробных, символов, ...); `arrayName` — название массива; `arraySizeRow` — количество одномерных массивов, `arraySizeColumn` — размер каждого одномерного массива. Количество одномерных массивов лежит в диапазоне $[0 \dots \text{arraySizeRow} - 1]$, размерность одномерных массивов — $[0 \dots \text{arraySizeColumn} - 1]$. Обращение к элементу двумерного массива по индексам — `arrayName[`*indexRow*`][`*indexColumn*`]`.

Инициализация двумерного массива:

- Использовать инициализатор {} при объявлении массива:

```
1 int example[][5] =
2 {
3     {100, 200, 300, 400},
4     {200, 600, 1000, 1400},
5     {1000, 800, 600, 400}
6 }
```

  Обратите внимание, что указывается размерность одномерных массивов, но не количество этих массивов!

- Инициализировать массив при помощи циклов:

```
1 const int rowSize = 3;
2 const int columnSize = 5;
3 int myArray[rowSize][columnSize];
4
5 for (int i = 0; i < rowSize; ++i)
6     for (int j = 0; j < columnSize; ++j)
7         myArray[i][j] = ...;
```

Пример инициализированного массива:

	0	1	2	3
0	24	25	23	22
1	1	2	3	4
2	3	5	7	11
3	10	15	20	25
4	11	22	33	44
5	0	2	4	6

2.5. Строки

2.5.1 Инициализация строк

Для включения строк C++ необходимо подключить заголовочный файл <string>. Сам класс находится в пространстве имен **std**. Объект string можно инициализировать:

1. С помощью конструктора:

```
1 std::string text("Hello");
2 std::string anotherText(3, 'p'); // "ppp"
3 std::string emptyString;
```

   В последнем случае std::string emptyString вызывается конструктор по умолчанию, который создает пустую строку

2. При помощи оператора =, который вызывает конструктор:

```
1 std::string month = "August";
2 std::string emptyString = "";
```

Числа нельзя присваивать напрямую строке:

```
1 std::string error = 'c'; // ERROR
2 std::string anotherError(8); // ERROR
```

Оператор >> потока std::cin может также работать со строками:

```
1 std::string userAnswer;
2 std::cin >> userAnswer;
```

2.5.2 Взятие символа строки ([], at()), размер строки (length(), empty())

После инициализации строки можно обращаться к отдельным символам строки:

1. Через оператор `[]`:

```
1 std::string student = "Vasya";
2 std::cout << student[0]; // 'V'
3 std::cout << student[4]; // 'a'
4
5 student[2] = 'a';
6 std::cout << student; // "Vaaya"
```

2. Через метод `at(int index)` класса `string`:

```
1 std::string student = "Vasya";
2 std::cout << student.at(0); // 'V'
3 std::cout << student.at(3); // 'y'
4
5 student.at(0) = 'M';
6 std::cout << student; // "Masya"
```

Индексы, по которым можно обращаться к элементам строки, лежат в диапазоне $[0 \dots \text{stringLength} - 1]$.

Размер строки можно узнать с помощью метода `length()`:

```
1 std::string myString = "abcd";
2 std::cout << myString.length(); // 4
```

Для того, чтобы узнать, является ли строка пустой, можно вызывать метод `empty()`:

```
1 std::string nonEmptyString("qew12314");
2 std::string emptyString;
3
4 std::cout << nonEmptyString.empty() << std::endl; // 0
5 std::cout << emptyString.empty() << std::endl; // 1
```

2.5.3 Присваивание (assign()) и конкатенация строк(append())

Возможные варианты присваивания:

```
1 std::string str1("test");
2 std::string str2,str3,str4;
3
4 str2 = str1; // str2 = "test";
5 str3.assign(str1); // str3 = "test";
```

Метод `assign` можно использовать, когда необходимо присвоить `подстроку`. В таком случае метод принимает дополнительные параметры:

`assign(source, startIndex, length)`

source — строка типа `string`, из которой будет браться подстрока, `startIndex` — начальный индекс, с которого будет начинаться копирование, `length` — количество символов, которые будут скопированы. Необходимо, чтобы выполнялось условие $length \leqslant source.length() - startIndex$.

Пример:

```
1 std::string str1 = "abcd";
2 std::string str2;
3 str2.assign(str1, 2, 2);
4 std::cout << str2; // str2 = "cd";
```

Возможные варианты конкатенации:

```
1 std::string str1("abcd");
2 std::string str2;
3
4 str2 += str1; // str2 = "abcd";
5 std::string str3(str1 + str2); // str5 = "abcdabcd"
6
7 std::string str4 = "zyx";
8 str4.append(str1); // str4 = "zyxabcd";
```

Метод `append` можно использовать, когда необходима конкатенация `подстроки`. В таком случае метод принимает дополнительные параметры:

`append(source, startIndex, length)`

`source` — строка типа `string`, из которой будет браться подстрока, `startIndex` — начальный индекс, с которого будет начинаться копирование, `length` — количество символов, которые будут скопированы. Необходимо, чтобы выполнялось условие $length \leqslant source.length() - startIndex$.

Пример:

```
1 std::string str1 = "abcdfg";
2 std::string str2 = "111";
3 str2.append(str1, 1, 2);
4
5 std::cout << str2; // str2 = "111bc"
```

2.5.4 Сравнение строк (compare())

Строки сравниваются лексикографически. Сравнение происходит по первому несовпадающему символу, в противном случае сравниваются длины строк. Строки можно сравнивать с помощью операторов ==, >, <:

```
1  std::string str1("abcd");
2  std::string str2("Hello");
3  std::string str3(str2);
4  std::string str4("zxc");
5  std::string str5("zxcd");
6
7  std::cout << (str1 == str2) << endl; // false (0)
8  std::cout << (str2 == str3) << endl; // true (1)
9  std::cout << (str1 < str4) << endl; // true (1)
10 std::cout << (str1 > str2) << endl; // true (1)
11 std::cout << (str4 > str5) << endl; // false (0)
```

Сравнивать строки между собой можно с помощью метода `compare()`. Функция возвращает 0, если строки одинаковы; значение меньше 0, если строка лексикографически меньше или все сопоставимые символы одинаковы, но строка меньше; значение больше 0, если строка лексикографически больше или все сопоставимые символы одинаковы, но строка больше.

Пример:

```
1  std::string str1("abcd");
2  std::string str2("Hello");
3  std::string str3("zxc");
4  std::string str4("abcdfff");
5
6  std::cout << str1.compare(str1) << endl; // 0
7  std::cout << str1.compare(str2) << endl; // 1
8  std::cout << str1.compare(str3) << endl; // -1
9  std::cout << str1.compare(str4) << endl; // -1
```

Метод `compare()` может сравнивать подстроку вызываемой строки с другой строкой:

```
1  compare(startIndex, length, anotherString)
```

`startIndex` — начальный индекс, с которого будет начинаться копирование, `length` — количество символов, которые будут скопированы, `anotherString` — стро-

ка, с которой будет сравниваться подстрока. Необходимо подбирать `startIndex` и `length` так, чтобы подстрока была корректной.

Пример:

```
1 std::string str1("abcdefgh");
2 std::string str2("abcd");
3
4 std::cout << str1.compare(0, 4, str2) << endl; // 0
5 std::cout << str2.compare(3, 1, str1) << endl; // 1
```

Метод `compare` может сравнивать подстроку вызываемой строки с подстрокой другой строки

```
1 compare(startIndex, length, anotherString, anotherStartIndex,
      anotherLength)
```

`startIndex` — начальный индекс, с которого будет начинаться копирование, `length` — количество символов, которые будут скопированы, `anotherString` — строка, с которой будет сравниваться подстрока, `anotherStartIndex` и `anotherLength` аналогичны по своим функциям с `startIndex` и `length`.

2.5.5 Подстрока (substr)

С помощью метода `substr` можно брать подстроки оригинальной строки:

`substr(startIndex, length)`

Пример использования:

```
1 std::string first("abcdefg");
2 std::string subFirst = first.substr(3, 4); // "defg"
```

2.5.6 Поиск подстрок и символов (find, rfind, find_first_of, find_last_of, find_first_not_of, find_last_not_of)

Для поиска подстрок необходимо использовать методы `find()`, который ведет поиск слева направо, и `rfind()`, который ищет справа налево. На вход методы принимают подстроку типа `std::string` и возвращают индекс начальной позиции этой подстроки. Если данной подстроки нет, то возвращается `std::string::npos`.

Примеры:

```
1 std::string str("abcdefghiabc");
2
3 std::cout << str.find("abc") << std::endl; // 0
```

```
4 std::cout << str.find("fg") << std::endl; // 5
5 std::cout << str.rfind("abc") << std::endl; // 9
6 std::cout << str.rfind("zzz") << std::endl; // string::npos
```

Методы `find_first_of()`, `find_last_of()`, `find_first_not_of()`, `find_last_not_of()` принимает на вход набор символов в виде `std::string` и возвращают индекс первого\последнего вхождения\отсутствия в зависимости от своей функции, в противном случае методы возвращают `std::string::npos`.

Назначения методов:

- `find_first_of` — возвращает индекс первого вхождения любого из символов параметра в строке;
- `find_last_of` — возвращает индекс последнего вхождения любого из символов параметра в строке;
- `find_first_not_of` — возвращает индекс первого вхождения любого символа **не** из символов параметра в строке;
- `find_last_not_of` — возвращает индекс последнего вхождения любого символа **не** из символов параметра в строке.

Примеры:

```
1 std::string str("abcdefghiabc");
2
3 std::cout << str.find_first_of("tree") << std::endl; // 4
4 std::cout << str.find_last_of("cat") << std::endl; // 11
5 std::cout << str.find_first_not_of("father") << std::endl; // 1
6 std::cout << str.find_last_not_of("car") << std::endl; // 10
```

2.5.7 Удаление символов из строки (`erase()`, `clear()`)

При помощи метода `erase()` можно удалить n–ое количество символов:

`erase(index, length)`

`index` указывает индекс с которого необходимо начать удаление, `length` указывает сколько символов необходимо удалить. Если не указывать параметр `length`, то будут удаление будет происходить до конца строки.

Примеры:

```
1 std::string test("abcdefgh");
2 test.erase(3);
3 std::cout << test << std::endl; // test = "abc"
4
5 test = "12345678";
6 test.erase(1, 6);
7 std::cout << test << std::endl; // test = "18";
```

Если необходимо полностью очистить строку, то необходимо использовать метод `clear()`:

```
1 std::string test("1234abcd");
2 std::cout << test.empty() << std::endl; // 0
3
4 test.clear();
5 std::cout << test.empty() << std::endl; // 1
```

2.5.8 Замена символов в строке (`replace()`)

Для замены символов в строке необходимо использовать метод `replace()`:

`replace(index, length, replaceString)`

`index` указывает с какого индекса в строке необходимо произвести замену, `length` указывает сколько символов необходимо заменить, `replaceString` является строкой, на которую происходит замена.

Пример:

```
1 std::string test("1234abcd");
2 test.replace(4, 4, "5678");
3 std::cout << test << std::endl;
4 // test = "12345678"
```

2.5.9 Вставка символов в строку(`insert()`)

Для вставки символов в строку необходимо использовать метод `insert()`:

`insert(index, insertString)`

`index` указывает индекс, с которого необходимо сделать вставку, `insertString` является строкой, которую необходимо вставить.

Пример:

```
1 std::string test("14");
2 test.insert(1, "23");
3 std::cout << test << std::endl; // "1234"
```

2.6. Функции

2.6.1 Синтаксис

В языке C++ можно создавать пользовательские функции, которые можно использовать в своих программах. Функции помогают структурировать код, делать его более понятным. Если в коде присутствуют одинаковые куски кода, то их имеет смысл вынести в функции.

Синтаксис пользовательских функций:

```
1 FunType functionName(ParamType1 paramName1, ..., ParamTypeN
     paramNameN)
2 {
3     ...
4 }
```

`FunType` — тип возвращаемого значения функции, `functionName` — наименование функции, `ParamType` — тип параметра, `paramName` — наименование параметра.

Пример функции, которая принимает в качестве параметра целочисленное число и возвращает его квадрат:

```
1 long long sqr(int value)
2 {
3     return (long long)value * value;
4 }
```

Функция может возвращать любой базовый тип в C++, любое пользовательское значение при помощи ключевого слова `return`. В таком случае произойдет выход из функции и передача значения. Также функция может ничего не возвращать, в таком случае необходимо в качестве типа использовать ключевое слово `void`. В случае, если использовать `return` в функции, которая возвращает значение `void`, то функция просто завершит свою работу и передаст управление программе дальше.

Пример функции, которая выводит строку на экран в том случае, если она не пустая:

```
1 void printString(const std::string &text)
2 {
3     if (!text.empty())
4         std::cout << text << std::endl;
5 }
```

2.6.2 Прототипы функций

Функция вместе с реализацией должна находиться раньше, чем происходит её вызов, в противном случае произойдет ошибка компиляции, так как компилятор не будет знать о существовании данной функции.

```
1  long long sqr(int value)
2  {
3      ...
4  }
5
6  int main()
7  {
8      ...
9      long long a = sqr(5);
10     ...
11     return 0;
12 }
```

Однако существует способ разделить объявление функции и её определение. Для этого необходимо использовать *прототипы функций*. До использования функции описывается её *сигнатура* — тип возвращаемого значения функции, имя функции и её параметры. Затем в любом другом месте модуля описывается *определение* функции — ее сигнатура и тело (код):

```
1  // declaration - signature
2  long long sqr(int value);
3
4  int main()
5  {
6      ...
7      long long a = sqr(5); // function call
8      ...
9      return 0;
10 }
11
12 // definition
13 long long sqr(int value)
14 {
15     return (long long)value * value;
16 }
```

2.6.3 Передача аргументов в функции по значению и по ссылке

Аргументы в функцию можно передавать по значению (в таком случае на стороне функции произойдет полное копирование аргумента, что при больших типах данных может замедлить работу, но зато изменение аргумента в функции не повлияет на переменную, которая была передана извне функции), либо по ссылке (изменение переменной будет произведено как и в самой функции, так и вне неё). Способ передачи аргумента задается в *сигнатуре* функции.

Аргумент по значению:

```
1  void incCopy(int value)
2  {
3      ++value;
4  }
5
6  int main()
7  {
8      int a = 1;
9      incCopy(a);
10     std::cout << a << std::endl; // a = 1;
11 }
```

Аргумент по ссылке:

```
1  void incRef(int &value)
2  {
3      ++value;
4  }
5
6  int main()
7  {
8      int a = 1;
9      incRef(a);
10     std::cout << a << std::endl; // a = 2;
11 }
```

2.6.4 Аргументы по умолчанию

У параметров пользовательских функций можно выставлять аргументы по умолчанию, которые будут применяться, если в аргументы функции ничего не будет передаваться при вызове. Данные аргументы должны быть самыми правыми аргументами в списке параметров функции. Если вызывается функция с двумя

или более аргументами по умолчанию и если пропущенный аргумент не является самым правым в списке аргументов, то все аргументы справа от пропущенного тоже пропускаются.

Аргументы по умолчанию обычно указываются в прототипе функции при помощи оператора =, в реализации функции указывать значения по умолчанию не нужно.

Пример функции, которая подсчитывает площадь прямоугольника, в том случае, если указана ширина, в противном случае подсчитывается площадь квадрата по переданной длине:

```
1  double getSquare(double height, double width = -1.0);
2
3  int main()
4  {
5      std::cout << getSquare(5.0, 6.0) << std::endl; // 30.0
6      std::cout << getSquare(10.0) << std::endl // 100.0
7  }
8
9  double getSquare(double height, double width)
10 {
11     return weight < 0 ? height * height : height * width;
12 }
```

2.6.5 Перегрузка функций

C++ позволяет определить несколько функций с одним и тем же именем, если эти функции имеют различные наборы параметров (различные в том, что касается типа или числа параметров, либо порядка следования типов). Эта возможность называется *перегрузкой функций*. Перегрузка обычно используется обычно для создания нескольких функций, выполняющий сходные задачи, но над различными типами данных. Перегруженные функции различаются по *сигнатурам*. Сигнатура — комбинация имени функции и типов её параметров (в определенном порядке).

Пример функций, которые возвращают квадрат числа:

```
1  int square(int value)
2  {
3      return value * value;
4  }
```

```
5
6  double square(double value)
7  {
8      return value * value;
9  }
10
11 int main()
12 {
13     std::cout << square(5) << std::endl; // 25
14     std::cout << square(7.5) << std::endl; // 56.25
15     return 0;
16 }
```

2.7. Структуры

2.7.1 Синтаксис объявления

Структуры — пользовательский тип данных, который позволяет группировать переменные с разными типами под одним именем. Этот новый тип данных практические не отличается от встроенных типов данных и его можно использовать в своей программе. Объявлять структуры необходимо в самом начале файла после объявления `#include` и использования пространств имен (`using namespace ...`), но до функций. Делается это для того, чтобы каждая функция могла «знать» о существовании данной структуры.

Синтаксис структуры:

```
1 struct structName
2 {
3     Type variableName1;
4     Type variableName2;
5     ...
6     Type variableNameN;
7 };
```

`structName` — пользовательское имя структуры, `Type` — тип переменной в структуре, `variableName` — имя переменной (оно же поле структуры).

Пример структуры:

```
1 struct Person
2 {
3     std::string name;
```

```
4       std::string sirname;
5       unsigned int age;
6   };
```

Создание экземпляра структуры: `structName ourStruct;`

2.7.2 Конструкторы структур

В структуре могут присутствовать функции, которые имеют полный доступ к полям структуры. Вызвать функцию можно через экземпляр структуры.

Пример функции:

```
1   struct simpleStructure
2   {
3       int a;
4       void incA()
5       {
6           ++a;
7       }
8   };
9
10  int main()
11  {
12      simpleStructure test;
13      test.incA();
14  }
```

Для структуры можно указывать специальную функцию — *конструктор*, с помощью которого при инициализации экземпляра структуры можно определять поля структуры:

```
1   struct Person
2   {
3       std::string name;
4       std::string surname;
5       unsigned int age;
6       // constructor
7       Person(std::string personName, std::string personSurname,
            unsigned int personAge)
8       {
9           name = personName;
10          sirname = personSurname;
11          age = personAge;
```

```
12      }
13  };
```

При объявлении конструктора нужно следовать следующим правилам:

- Конструктор должен иметь такое же имя, что и название структуры;
- Перед конструктором не должно быть типа (даже `void`'а).

Так как конструктор является функцией, то оно может принимать на вход параметры, с помощью которых можно инициализировать поля структуры. Также структура может иметь несколько конструкторов с разными параметрами.

Так же структура может иметь конструктор по умолчанию, который вызывается, если не был вызван ни один другой конструктор. Конструкторы по умолчанию используются для того, чтобы поля не были инициализированы «мусором».

Пример конструктора по умолчанию:

```
1  struct Person
2  {
3      std::string name;
4      std::string surname;
5      unsigned int age;
6
7      // Default constructor
8      Person()
9      {
10         name = "";
11         surname = "";
12         age = 0;
13     }
14 };
```

При инициализации экземпляра структуры необходимо передать параметры в конструктор:

```
1  int main()
2  {
3      ...
4      Person man("Alexander", "Popov", 156);
5      ...
6  }
```

Если передавать на данный момент времени нечего, а необходимы корректно определенные поля, то необходимо использовать конструктор по умолчанию:

```
1 int main()
2 {
3     ...
4     Person man();
5     ...
6 }
```

Также имеется возможность просто объявить экземпляр структуры. Если конструктор по умолчанию объявлен, то он так же как и в прошлом примере будет вызван, в противном случае поля структуры останутся неинициализированными.

```
1 int main()
2 {
3     ...
4     Person man;
5     ...
6 }
```

2.7.3 Обращение к полям структуры

После того, как экземпляр структуры создан, можно обращаться к его полям и изменять их. Чтобы получить доступ к полю, необходимо использовать оператор . к экземпляру структуры:

```
1  int main()
2  {
3      Person man();
4
5      std::string name = "";
6      std::string surname = "";
7      unsigned int age = 0;
8
9      std::cout << "Please, enter your name:" << std::endl;
10     std::cin >> name;
11
12     man.name = name;
13
14     std::cout << "Please, enter your surname:" << std::endl;
15     std::cin >> surname;
16
17     man.surname = surname;
18
```

```
19      std::cout << "Please, enter your age: " << std::endl;
20      std::cin >> age;
21
22      man.age = age;
23
24      std::cout << "Hello, dear " << man.name << "! Your surname
           is " << man.surname << " and your age is " << man.age <<
           std::endl;
25  }
```

2.7.4 Пример использования структур

В данном примере выводится максимальная площадь среди 10 прямоугольников. Для представления прямоугольника используется структура `Rectangle`, в которой есть переменные, отвечающие за высоту и ширину прямоугольника, есть конструктор по умолчанию и функция, которая возвращает площадь прямоугольника:

```
1  #include <iostream>
2  #include <cstdlib>
3  #include <ctime>
4
5  struct Rectangle
6  {
7      unsigned int width, height;
8
9      Rectangle()
10     {
11         width = 0;
12         height = 0;
13     }
14
15     unsigned int calcSquare()
16     {
17         return width * height;
18     }
19
20 };
21
22 int main()
23 {
```

```
24      srand(time(NULL));
25
26      const int size = 10;
27      Rectangle recArray[size];
28
29      for (int i = 0; i < size; ++i)
30      {
31          recArray[i].height = rand() % 100 + 1;
32          recArray[i].width = rand() % 100 + 1;
33      }
34
35      int maxSquare = recArray[0].calcSquare();
36
37      for (int i = 0; i < size; ++i)
38      {
39          int recSquare = recArray[i].calcSquare();
40
41          std::cout << "Rectangle " << i << ": height = " <<
                recArray[i].height << ", width = " <<
                recArray[i].width << ", square = " << recSquare <<
                std::endl;
42          if (maxSquare < recSquare)
43                  maxSquare = recSquare;
44      }
45
46      std::cout << "Maximum square is " << maxSquare << std::endl;
47
48      return 0;
49  }
```

Для генерации случайных чисел используется функция `rand()`, которая генерирует псевдослучайное число. При помощи оператора `% value` можно задавать диапазон генерации чисел: от 0 до `value`. При помощи оператора `+` также можно поправить диапазон, прибавляя к результату взятия остатка некое число. Для использования данной функции необходимо использовать библиотеку `cstdlib`. Для того, чтобы при функция `rand()` работала корректно, то необходимо задать специальное число (*seed*) для инициализатора генератора случайных чисел `srand()`. В качестве числа берется текущая дата, представленная в виде количества секунд — `time(NULL)`. Для функции `time(NULL)` необходимо использовать библиотека `ctime`.

На 27 строке задается массив из 10 структур `Rectangle`. В данном случае для структур необходим конструктор по умолчанию. В первом цикле происходит присвоение высот и широт для прямоугольников при помощи функции `rand()`. Во втором цикле идет перебор площадей прямоугольников и выбор самого максимального.

3. ООП (Основы)

3.1. Классы

3.1.1 Определение, объекты

Объектно-ориентированное программирование (ООП) — парадигма программирования, в которой основными концепциями являются понятия объектов и классов. Если проводить аналогии с объектами из реальной жизни, то машина — это *объект*, который построен по неким чертежам или схемам (*класс*), которые описывают данный объект — его характеристики и возможности. У машины может быть свой вес, цвет, вместимость — всё это является *свойствами* объектов. Значения этих свойств у различных машин (объектов) может различаться. Возможность ездить, подавать сигнал — это *методы* объекта. Имея один чертеж (класс), можно создавать огромное количество машин (объектов), у которых будут в наличии все эти методы, причем не нужно будет задумываться о реализации этих методов для каждого из объектов!

Основные принципы ООП:

- Инкапсуляция — данное свойство позволяет объединить данные и методы, работающие с ними, в классе, и скрыть детали реализации от пользователя;
- Наследование — свойство, позволяющее создать новый класс-потомок на основе уже существующего, при этом все характеристики класса родителя присваиваются классу-потомку;
- Полиморфизм — свойство классов, позволяющее использовать объекты классов с одинаковым интерфейсом без информации о типе и внутренней структуре объекта;

В C++ класс — это пользовательский тип данных, который может группировать в себе как переменные различных типов — *свойства*, так и различные функции — *методы*.

Класс можно объявить через ключевое слово `class` и в фигурных скобках перечислить его свойства и методы:

```
1 class Message
2 {
3 public:
```

```
4       void showMessage(const std::string& text)
5       {
6           std::cout << "Hello user! " << text << std::endl;
7       }
8   };
9
10  int main()
11  {
12      Message worldMessage;
13      worldMessage.showMessage("Welcome to our world!"); // "Hello
            user! Welcome to our world!"
14      return 0;
15  }
```

В данном примере рассматривается класс `Message`, который имеет метод `showMessage`, который выводит на консоль текст, состоящий из строки `"Hello user!"` и параметра `text`, которые соединены между собой. В функции `main` объявляется *объект* `worldMessage` класса `Message`, у которого вызывается метод `showMessage`.

Ещё один пример класса:

```
1   class Course
2   {
3   private:
4       std::string courseName;
5
6   public:
7       Course(const std::string& course)
8       {
9           courseName = course;
10      }
11
12      void printCourseName()
13      {
14          std::cout << "Your course is " << courseName;
15      }
16
17      std::string& getCourseName()
18      {
19          return courseName;
20      }
21  };
22
```

```
23 int main()
24 {
25     Course firstCourse("C++");
26     Course secondCourse("Geometry");
27
28     firstCourse.printCourseName(); // "Your course is C++"
29     secondCourse.printCourseName(); // "Your course is Geometry"
30
31     std::cout << "My favorite course is " <<
          firstCourse.getCourseName() << " and not so favorite
          course is " << secondCourse.getCourseName() << std::endl;
32
33     // My favorite course is C++ and not so favorite course is
          Geometry
34
35     return 0;
36 }
```

Данный класс `Course` представляет собой предмет в школе. У него есть свойство `courseName`, которое хранит в себе название курса; метод `getCourseName`, который возвращает наименование курса; `printCourseName` печатает название курса; `Course` является *конструктором* класса (подробнее см. 3.1.4) и инициализирует `courseName` параметром. В функции `main` создается два курса `firstCourse`, который инициализируется строкой *C++* и `secondCourse`, который инициализируется строкой *Geometry*. Дальше выводятся названия этих функций при помощи метода `printCourseName` и берутся напрямую названия курсов через `getCourseName` для создания строки.

3.1.2 Модификаторы доступа к членам класса

В примере с классом `Message` можно увидеть ключевое слово `public`, которое указывает на модификатор доступа к этому методу. Всего существует 3 модификатора доступа в `C++`:

- `public` — все свойства и методы открыты для всех, т.е. через объект можно будет обращаться к методам, или изменять свойства (изменения могут происходить как внутри класса, так и снаружи);

- `private` — свойства и методы в данной секции можно использовать только внутри класса, т.е. свойства изменяются только внутри *этого* класса и методы можно вызывать только внутри *этого* класса;

- `protected` — доступ к свойствам и методам открыт как и внутри класса, так и внутри производных классов от изначального, снаружи свойства не изменяются, методы не вызываются;

Правила объявления секций доступа:

```
1  class Test
2  {
3      public:
4          // public properties and methods
5          int publicInt;
6          void incPublicInt()
7          {
8              ++publicInt;
9          }
10
11     private:
12         // private properties and methods
13         int privateInt;
14         void showMessage()
15         {
16             std::cout << "you can't invoke me from object!";
17         }
18
19     protected:
20         // protected properties and methods
21         // ...
22 };
```

Свойство `publicInt` можно изменять как и внутри класса, так и снаружи через объект данного класса. Метод `incPublicInt` также можно использовать как внутри класса, так и снаружи через объект. Доступ к свойству `privateInt` и методу `showMessage` через объект данного класса запрещен. Модификатор `protected` необходим для ситуаций, когда происходит наследование классов.

3.1.3 Get- и Set- функции

Как правило, свойства в классах должны находится в `private` секциях доступа. Однако существует возможность создать функции, которые позволяют получать значение (`get`) и изменять эти значения (`set`) через объекты классов.

Метод `get` должна возвращать свойство класса:

```
1  class BankAccount
2  {
3  private:
4      double cash;
5  public:
6      double getCash()
7      {
8          return cash;
9      }
10 };
```

Метод `set` присваивает определённому свойству значение параметра, переданного в метод. При этом можно осуществлять проверку значения параметра, чтобы не произошло некорректной работы в дальнейшем:

```
1  class BankAccount
2  {
3  private:
4      double cash;
5  public:
6      void setCash(double newCash)
7      {
8          if (newCash >= 0)
9              cash = newCash;
10     }
11 };
```

3.1.4 Конструктор и деструктор

Конструктор — функция, которая вызывается в самом начале «жизни» объекта класса (сразу же после инициализации объекта в памяти) и необходима для подготовки объекта к работе: инициализация свойств и вызовов методов. *Деструктор* — функция, которая вызывается перед тем, как будет удален объект

из памяти и предназначен прежде всего для уничтожения внутренних данных, которые выделены в динамической памяти.

Конструктор должен начинаться с имени класса и у него не должно быть никакого типа перед названием. Конструктор обязательно должен иметь модификатор доступа `public`. Как и для обычной функции, в конструктор можно передавать различные параметры.

Пример конструктора с параметрами:

```
1  class Figure
2  {
3  private:
4      double square;
5
6  public:
7      Figure(double figureSquare) // constructor
8      {
9          square = figureSquare;
10     }
11 };
12
13 int main()
14 {
15     Figure circle(15.0); // circle.square = 15.0;
16     ...
17
18     return 0;
19 }
```

Также существует *конструктор по умолчанию*. Таким конструктором является всякий конструктор, у которого нет параметров и вызывается он в том случае, если создается объект без параметров. Если он явно не указан, то он генерируется автоматически с пустым содержимым.

```
1  class Figure
2  {
3  private:
4      double square;
5
6  public:
7      Figure() // default constructor
8      {
9          square = 0;
```

```
10      }
11
12      Figure(double figureSquare) // constructor
13      {
14          square = figureSquare;
15      }
16  };
17
18  int main()
19  {
20      Figure circle(15.0); // circle.square = 15.0;
21      Figure triangle; // triangle.square = 0.0;
22      ...
23
24      return 0;
25  }
```

Деструктор объявляется практически также, как и конструктор, но к названию добавляется символ ~:

```
1  class Figure
2  {
3  public:
4      ~Figure() // destructor
5      {
6          ...
7      }
8  };
```

3.1.5 Разделение класса на два файла. Отделения интерфейса от реализации

В языке C++ существует возможность разделить класс на *интерфейс* и *реализацию*. Интерфейс — описание свойств и методов класса без раскрытия деталей реализации. Реализация — это собственно программный код, которые реализует поставленные методы класса.

Интерфейс класса необходимо указывать в файле `className.h`, где *className* — название класса. В данном файле можно указать заголовочные файлы, которыми будет пользоваться класс; указать используемые пространства имен, а также ука-

зать сам класс с его свойствами и *прототипами методов*, которые включают в себя имена методов, списки параметров методов и их возвращаемые значения.

Пример интерфейса класса *Student*:

student.h

```
1  #ifndef STUDENT_H
2  #define STUDENT_H
3
4  #include<string>
5
6  using namespace std;
7
8  class Student
9  {
10 private:
11     string studentName;
12     string studentSurname;
13     int rating;
14     int ratingSize;
15
16 public:
17     Student(string name, string surname);
18
19     string getStudentName();
20     void setStudentName(const string& name);
21
22     string getStudentSurname();
23     void setStudentSurname(const string& surname);
24
25     void addMark(int mark);
26     double getAverageRating();
27
28     void printInformation();
29 };
30
31 #endif
```

В начале заголовочного файла ставится защита подключения — *include guard*:

```
1  #ifndef CLASSNAME_H
2  #define CLASSNAME_H
3  ...
4  #endif
```

, где вместо `CLASSNAME` — название класса. Данный механизм используется для того, чтобы в коде при многочисленном обращении к данному заголовочному файлу оставался один экзмепляр данного файла, в противном случае возникнет ошибка компилятора по поводу дублирования файлов. Внутри данной конструкции необходимо перечислить все то, что необходимо защитить от копирования, в том числе и интерфейс классов.

Обратите внимание, что в заголовочном классе нет реализации методов класса, вместо них указаны только их прототипы.

В данном примере рассматривается класс `Student`. У него есть свойства `studentName` (имя студента), `studentSurname` (фамилия студента), `rating` (сумма баллов), `ratingSize` (количество введеных баллов). Для класса существует конструктор `Student`, который принимает на вход имя и фамилию студента; `get` и `set` методы для свойств `studentName` и `studentSurname`; метод `addMark`, который добавляет оценку к свойству `rating`; метод `getAverageRating`, который выводит средний балл студента и метод `printInformation`, который выводит всю информацию о студенте.

Реализацию класса необходимо указывать в файле `className.cpp`, где *className* — название класса. В нем необходимо указать необходимые заголовочные файлы, пространства имен, а также **обязательно** указать заголовочный файл, в котором находится интерфейс данного класса.

Пример реализации класса *Student*:

student.cpp

```
1  #include "student.h"
2  #include <iostream>
3
4  Student::Student(string name, string surname)
5  {
6      studentName = name;
7      studentSurname = surname;
8      rating = 0;
9      ratingSize = 0;
10 }
11
12 string Student::getStudentName()
13 {
14     return studentName;
15 }
16
```

```
17 void Student::setStudentName(const string& name)
18 {
19     studentName = name;
20 }
21
22 string Student::getStudentSurname()
23 {
24     return studentSurname;
25 }
26
27 void Student::setStudentSurname(const string& surname)
28 {
29     studentSurname = surname;
30 }
31
32 void Student::addMark(int mark)
33 {
34     if (mark > 0)
35     {
36         rating += mark;
37         ++ratingSize;
38     }
39 }
40
41 double Student::getAverageRating()
42 {
43     return (double)(rating) / ratingSize;
44 }
45
46 void Student::printInformation()
47 {
48     cout << "Student " << studentSurname << " " << studentName
          << " has average rating: " << getAverageRating() << endl;
49 }
```

Каждое имя метода предваряется именем класса и ::, *бинарной (двухместной) операцией разрешения области действия*. Таким образом каждая реализация методов «привязывается» к соответствующему методу к интерфейсу. Если не указывать ::, то связывания не произойдет!

Для того, чтобы использовать данный разделенный класс в своих программах, также необходимо указать заголовочный файл с интерфейсом класса:

```
#include "student.h"

int main()
{
    Student firstStudent("Mark", "Ivanov");

    firstStudent.addMark(4);
    firstStudent.addMark(5);
    firstStudent.addMark(5);

    firstSrudent.printInformation();

    return 0;
}
```

3.2. Наследование

3.2.1 Введение

Наследование — это способ повторного использования программного обеспечения, при котором новые *производные* классы (наследники) создаются на базе уже существующих *базовых* классов (родителей).

При создании новый класс является наследником свойств и методов ранее определенного базового класса. Создаваемый путем наследования класс является *производным* (derived class), который в свою очередь может выступать в качестве *базового* класса (based class) для создаваемых классов. Если имена методов производного и базового классов совпадают, то методы производного класса перегружают методы базового класса.

Наследование производится следующим образом:

```
class DerivedClassName : typeInheritance BaseClassName
{
public:
    ...
private:
    ...
protected:
    ...
};
```

, где `DerivedClassName` — производный класс, `typeInheritance` — способ наследования, `BaseClassName` — базовый класс.

Пример наследования:

```
1  class Human
2  {
3  public:
4      std::string name;
5      unsigned int age;
6
7      Human(std::string humanName, unsigned int humanAge)
8      {
9          name = humanName;
10         age = humanAge;
11     }
12 };
13
14 class Student : public Human
15 {
16 public:
17     std::string university;
18
19     Student(std::string humanName, unsigned int humanAge,
           std::string studentUniversity) : Human(humanName,
           humanAge)
20     {
21         university = studentUniversity;
22     }
23 };
24
25 int main()
26 {
27     Student student("Anton", 23, "NOVSU");
28     std::cout << "Student " << student.name << ", age " <<
           student.age << ", studies in " << student.university <<
           std::endl;
29     // Student Anton, age 23, studies in NOVSU
30     return 0;
31 }
```

В данном примере рассмотрен класс `Human`, у которого есть свойства `name` (имя человека) и `age` (возраст). При помощи конструктора можно инициализировать

эти свойства. Класс `Student` наследуется от класса `Human`, в результате чего также владеет свойствами `name` и `age`, а также добавляет своё свойство `university` (университет, где он учится). Каждый базовый класс, который участвует в наследовании, должен быть инициализирован во время инициализации своего потомка. Поэтому в конструктор класса `Student` передаются параметры для класса `Human`, и данные параметры используется в конструкторе класса `Human`.

3.2.2 Модификатор доступа protected

Если в базовом классе свойства и методы имеют модификатор доступа `private`, то производный класс не будет иметь доступ к ним при наследовании. Если свойства и методы будут иметь модификатор доступа `public`, то доступ будут иметь абсолютно все, что не есть хорошо. При помощи модификатора `protected` можно разрешить производному классу обращаться к свойствам и методам базового класса, но при этому доступ извне к ним не будет осуществляться!

Пример:

```
1  class Rectangle
2  {
3  protected:
4      double width, length;
5  public:
6      Rectangle(double recWidth, double recLength)
7      {
8          width = recWidth;
9          length = recLength;
10     }
11
12     double getSquare()
13     {
14         return width * length;
15     }
16 };
17
18 class Cuboid : public Rectangle
19 {
20 private:
21     double height;
22 public:
```

```
23      Cuboid(double cubWidth, double cubLength, double cubHeight)
            : Rectangle(cubWidth, cubLength)
24      {
25          height = cubHeight;
26      }
27
28      double getVolume()
29      {
30          return width * length * height;
31      }
32  };
33
34  int main()
35  {
36      Rectangle rectangle(5.0, 3.0);
37      Cuboid cuboid(10.0, 6.0, 2.0);
38
39      std::cout << "Rectangle's square = " <<
            rectangle.getSquare() << std::endl;
40      std::cout << "Cuboid's bottom square = " <<
            cuboid.getSquare() << ", cuboid's volume = " <<
            cuboid.getVolume() << std::endl;
41
42      // Rectangle's square = 15
43      // Cuboid's bottom square = 60, cuboid's volume = 120
44
45      return 0;
46  }
```

В данном примере рассматриваются классы `Rectangle` (прямоугольник) и `Cuboid` (прямоугольный параллелепипед). У класса `Rectangle` есть свойства `width` (ширина) и `length` (длина), которые имею модификатор доступа `protected`, и конструктор, в который передаются значения ширины и высоты, а также публичный метод `getSquare`, который позволяет получить площадь прямоугольника. Класс `Cuboid` наследуется от класса `Rectangle`, имеет доступ к свойствами `width` и `length`, добавляет своё свойство `height` (высота) и новый метод `getVolume`, позволяющий получить объем. В методе `getVolume` класса `Cuboid` возвращает произведение длины, ширины и высоты, доступ к длине и ширине базового класса обеспечивается модификатором доступа `protected`.

3.2.3 Типы наследования

При наследовании классов можно указать тип наследования, которые могут модифицировать существующие модификаторы доступа. Следующая табличка раскрывает взаимодействие типов наследования и модификаторов доступа:

	Исходный модификатор доступа		
	`public`	`private`	`protected`
public-наследование	*public*	*private*	*protected*
private-наследование	*private*	*private*	*private*
protected-наследование	*protected*	*private*	*protected*