

Основы программирования

Практическая работа 2

Тема: Арифметические операции и преобразования

Цель работы

Целями данной работы являются изучение базовых арифметических операторов и операций и преобразования типов данных.

Краткие теоретические сведения

Традиционный ввод-вывод

Для использования традиционного ввода-вывода в C++, в программу необходимо включить заголовочный файл `<stdio.h>`(или `<cstdio>`).

Библиотека *stdio* предоставляет необходимый набор функций для ввода и вывода информации как в текстовом, так и в двоичном представлении.

Пример вывода:

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

Пример ввода:

```
int main()
{
    int a;
    scanf_s("%d", &a); // получаем целое число
    return 0;
}
```

Для осуществления ввода-вывода значений переменных используется множество различных спецификаторов, которые записываются в управляющей строке функции *scanf_s* или *printf*. Начинается спецификатор с символа «%» и далее следует символ обозначающий тип вводимого или выводимого значения.

Ввод-вывод с помощью потоков STL

Для использования консольного ввода-вывода с помощью потоков STL в программу необходимо включить заголовочный файл `<iostream>`.

Для выполнения операций ввода-вывода переопределены две операции:

>> – получить из входного потока;

<< – поместить в выходной поток.

Пример:

```
#include <iostream>
int main() {
    int a;
    std::cin >> a;
```

```

        std::cout << "A = " << a << std::endl;
        return 0;
    }

```

Перед каждым оператором ввода-вывода нужно указывать имя пространства имен «**std::**».

Чтобы можно было опустить указание пространства имен необходимо добавить строку:

```

using namespace std;
int main() {
    cout << "Hello, world!\n";
    int a;
    cin >> a;
    return 0;
}

```

Манипуляторы потока

Для работы большинства манипуляторов необходимо подключить библиотеку `<iomanip>`.

Рассмотрим некоторые манипуляторы:

Манипулятор	Описание
<code>endl</code>	Помещение в выходной поток символа конца строки '\n'
<code>boolalpha</code>	Вывод логических величин в текстовом виде (<i>true</i> , <i>false</i>)
<code>dec</code>	Установка основания 10-ой системы счисления
<code>oct</code>	Установка основания 8-ой системы счисления
<code>hex</code>	Установка основания 16-ой системы счисления
<code>setw(ширина)</code>	Устанавливает ширину поля вывода
<code>setfill('символ')</code>	Заполняет пустые места значением символа
<code>setprecision(точность)</code>	Устанавливает количество значащих цифр в числе (или после запятой) в зависимости от использования <code>fixed</code>
<code>fixed</code>	Вывод чисел с плавающей точкой в фиксированной форме
<code>showpos</code>	Показывает знак + для положительных чисел перед числом
<code>scientific</code>	Выводит число в экспоненциальной форме
<code>right / left</code>	Выравнивание по правой (по умолчанию) или левой границе. Сначала необходимо установить ширину поля, чтобы было пространство для выравнивания.

Пример:

```

#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    int a = 543;
    float b = 12.345;
    cout << setfill('-') << setw(10) << '-' << endl;
    cout << setfill('.') << setw(8) << left << a << endl;
    cout << setprecision(2) << fixed << b << endl;
    return 0;
}

```

```

-----
543.....
12.35

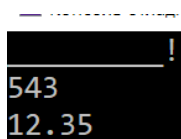
```

Вместо использования манипуляторов можно использовать методы потоков. Они работают и **без** подключения библиотеки *<iomanip>*. К примеру:

Метод	Описание
width(ширина)	Устанавливает ширину поля вывода
precision()	Устанавливает количество значащих цифр в числе
fill('символ')	Заполняет пустые места значением символа
get()	Ожидает ввода символа
getline(указатель, количество)	Ожидает ввода строки символов. Максимальное количество символов ограничено полем количество

Пример:

```
#include <iostream>
using namespace std;
int main()
{
    int a = 543;
    float b = 12.345;
    cout.width(10);
    cout.precision(2);
    cout.fill('_');
    cout << '!' << endl;
    cout << a << endl;
    cout << fixed << b << endl;
    return 0;
}
```



```

!
543
12.35

```

Операции

Язык C\C++ содержит большое количество встроенных операций. Для вызова выполнения операций используются **операторы**.

Оператор присваивания

Оператор присваивания (=) – специальный оператор, который помещает значение *правого* операнда в ячейку памяти (переменную), указанную как *левый* операнд. Общая форма:

имя_переменной = выражение;

Пример:

```
int a;
a = 1;
```

Так как оператор присваивания возвращает результат, то он может присутствовать в любом выражении языка C\C++.

Арифметические операции

Арифметические операции в C/C++:

Оператор	Операция
-	Вычитание, так же унарный минус
+	Сложение
*	Умножение
/	Деление. Если операция / применяется к целому типу, то остаток от деления отбрасывается, то есть результат также целое число.
%	Остаток от деления
--	Декремент, или уменьшение
++	Инкремент, или увеличение

Примеры:

```
int a = 10, b = 6, sum, ost, del;
sum = a + b; // результат 16
ost = a % b; // результат 4 - остаток от деления
del = a / b; // результат 1, так как дробная часть отбрасывается
```

Инкремент (++) увеличивает содержимое любой переменной на единицу и перезаписывает значение переменной.

Декремент (--) уменьшает содержимое любой переменной на единицу и перезаписывает значение переменной.

У обоих операторов есть 2 формы записи:

- Префиксная форма: *оператор операнд*
- Постфиксная форма: *операнд оператор*

Пример:

```
int a = 0, b = 2;
++a; // a станет 1
b--; // b станет 1
```

Однако эти формы отличаются при использовании их в выражениях. Пример:

```
int a = 0, res;
res = ++a; // a = 1, res = 1
int a = 1, res;
res = a--; // a = 0, res = 1
```

Приоритет выполнения арифметических операторов следующий:

Высший приоритет
Префиксная форма инкремента (++) и декремента (--)
Унарный минус (-)
Умножение (*), Деление (/), Деление по модулю (%)
Сложение (+), Вычитание (-)
Оператор присваивания (=)
Постфиксная форма инкремента (++) и декремента (--)
Низший приоритет

Составные операторы присваивания

Составное присваивание – это разновидность оператора *присваивания*, в которой запись сокращается и становится более удобной в написании. Например, оператор

```
x = x+10;
```

можно записать как:

```
x += 10;
```

Эти операторы существуют для всех бинарных арифметических операций:

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

Преобразование типов

Напомним, что C++ является *статически типизированным* языком программирования. То есть, если мы определили для переменной какой-то тип данных, то в последующем мы этот тип изменить не сможем. Однако нередко возникает необходимость присвоить переменной значения других типов. В этом случае применяется **преобразование типов**.

Безопасные преобразования

Те преобразования, при которых не происходит потеря информации, являются *безопасными*. Расширяющие преобразования можно считать *безопасными*.

В частности, можно выделить следующие цепочки преобразований:

```
bool → char → short → int → double → long double
```

```
int → long → long long
```

```
unsigned char → unsigned short → unsigned int → unsigned long
```

```
float → double → long double
```

Опасные преобразования

При опасных преобразованиях мы потенциально можем потерять точность данных. Как правило, это преобразования от типа с большей разрядностью к типу с меньшей разрядностью.

```
char letter = 295; // число 295 будет сокращено до 39
```

Несмотря на то, что некоторые типы имеют одинаковый размер, в них помещается разный диапазон значений. К примеру, если целочисленной переменной присваивается дробное число, то дробная часть после запятой отбрасывается.

```
int a = 3.4; // 3
```

Если же вещественной переменной присваивается целое число, и если целое число содержит больше бит информации, чем может вместить тип вещественной переменной, то часть информации усекается.

```
double b = 3500500000033; // 3.5005e+012 (33 в конце сокращается)
```

Классификация по способу осуществления преобразования

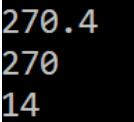
- *Неявное преобразование*. Это автоматический вид преобразования.
- *Явное преобразование (приведение типов)*. Преобразование производится программистом, тогда, когда это необходимо.

Ряд преобразований компилятор может производить **неявно**, то есть *автоматически*. Это те преобразования, которые мы рассмотрели в предыдущих примерах.

Но иногда нам нужно **явно** указать к какому типу данных мы хотим преобразовать значение. Для такой цели используется **оператор приведения типа**.

Явное преобразование типов данных выполняется с помощью *оператора ()*:

```
double z = 270.4;
float x = (int) z;
float y = char (z);
cout << z << "\n" << x << "\n" << y;
```



```
270.4
270
14
```

Преобразование типов в выражении

Предположим у нас есть следующие переменные:

```
int I = 27;
short S = 2;
float F = 22.3;
bool B = false;
```

Составим такое выражение:

$I - F + S * B$

Какого типа будет результат? Решить это просто, если представить выражение в виде типов данных:

$\text{int} - \text{float} + \text{short} * \text{bool}$

Если в каком-либо выражении используются разные типы данных, то результат, приводится к **большему** из этих типов.

Таким образом, результат выражения будет иметь тип **float**.

Постановка задания

Напишите программы для решения следующих задач. При решении задач для ввода-вывода используйте **потоки STL**. Также для форматирования ввода вывода используйте соответствующие манипуляторы.

Задачи:

- 1) За один день хомячок съедает K грамм корма. Написать программу, определяющую закупку корма в килограммах на N дней. Здесь K и N являются переменными, значение которых вводится. Пример:

```
Входными данными являются:
Расход корма за день (грамм) - 20
Количество дней - 30
-----
Выходные данные:
Необходимый объем корма - 0.6 кг
```

- 2) Пользователь вводит с клавиатуры время в секундах. Необходимо написать программу, которая переведет введенные пользователем секунды в формат «часы, минуты, секунды» и выведет их на экран.

```
Секунды: 5000
Время: 1 час 23 минуты 20 секунд
```

- 3) Пользователь вводит два вещественных числа. Посчитайте отдельно сумму их целых и дробных частей.

```
Число 1: 10.5
Число 2: 11.67
-----
Сумма целых: 21
Сумма дробных: 1.17
```