ГОСУДАРСТВЕННЫЙ КОМИТЕТ СВЯЗИ, ИНФОРМАТИЗАЦИЙ И
ТЕЛЕКОММУНИКАЦИОННЫХ ТЕХНОЛОГИЙ
РЕСПУБЛИКИ УЗБЕКИСТАН

НУКУССКИЙ ФИЛИАЛ
ТАШКЕНТСКОГО УНИВЕРСИТЕТА ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

Курсовая работа

По предмету: Программирование на C++
На тему: Структуры на языке C++

Студента 2в курса направления «Телекоммуникационные технологии»

Выполнил Балтабаев А.

Преподаватель: Ядгаров Ш.

Нукус 2014

План

Введение

Теоретическая часть

1. Определение структур

1.1. Структура программы на языке С++

1.2. Стандарты языка С++

1.3. Представление данных в языке С++

1.4. Оператор присваивания

1.5. Системы счисления

1.6. Арифметические операции

1.7. Поразрядные операции языка С++

1.8. Директивы препроцессора

Практическая часть

Заключение

Список использованной литературы

Введение

Лучший способ начать изучать язык программирования - это посмотреть пример простой программы и понять ее структуру. Поэтому в начале первой главы приводится пример программы на C++ и описание ее структуры. Затем рассматриваются возможные типы данных, правила определения переменных разного типа и программирование арифметических операций включая поразрядные операции над числами. Разработчики C++ развили концепцию структуры до класса.

Как и классы, структуры C++ могут содержать спецификаторы доступа, элемент-функции, конструкторы и деструкторы. На самом деле единственным отличием структур от классов в C++ является то, что элементы структуры по умолчанию имеют доступ public, если для них не указано никакого спецификатора.

1. Определение структур

Структуры являются агрегатными типами данных — другими словами, они могут строиться из элементов различного типа, в том числе других структур. Рассмотрим следующее определение структуры:

```
struct Card
{
char *face;
char *suit;
}; // конец struct Card
```

Ключевое слово struct начинает определение структуры Card. Идентификатор Card является в C++ именем структуры и может использоваться для объявления переменных структурного типа (в C именем вышеприведенной структуры является struct Card). В данном примере Card является структурным типом. Данные (и, возможно, функции — точно так же, как в классах), объявленные внутри фигурных скобок определения структуры, являются ее элементами. Элементы одной и той же структуры должны иметь уникальные имена, но две различных структуры могут содержать элементы с одним и тем же именем, не вызывая конфликта. Каждое определение структуры оканчивается точкой с запятой.

Типичная ошибка программирования:
Пропуск точки с запятой, оканчивающей определение структуры, является синтаксической ошибкой.

Определение Card содержит два элемента типа char * с именами face и suit. Элементы структуры могут быть переменными основных типов данных (т.е. int, double и т.п.) или агрегатами, такими, как массивы, другие структуры и классы. Элементы данных одной структуры могут принадлежать к различным типам. Например, структура Employee могла бы содержать в качестве элементов символьные строки для имени и фамилии, элемент int для возраста служащего, элемент char, содержащий 'M' или 'F', для пола, элемент double для почасовой оплаты и т.д.

Структура не может содержать представителей себя самой. Например, в определении структуры Card нельзя объявить переменную структурного типа Card. Однако в структуру может быть включен указатель на Card. Структура, содержащая указатель на тот же самый структурный тип, называется автореферентной структурой.

Определение структуры Card не резервирует для нее какого-то места в памяти; оно только создает новый тип данных, используемый для объявления структурных переменных. Структурные переменные объявляются так же, как переменные других типов. Так, операторы

```
Card oneCard;
Card deck[ 52 ];
Card *cardPtr;
```

объявляют oneCard — одиночную структурную переменную типа Card, deck — массив из 52 элементов типа Card, и cardPtr — указатель на структуру Card. Переменные структурного типа могут объявляться также в разделенном запятыми списке, расположенном между закрывающей фигурной скобкой определения данной структуры и точкой с запятой, завершающей определение структуры. Например, предыдущие объявления могли бы быть включены в определение структуры Card:

```
struct Card
{
char *face;
char *suit;
} oneCard, deck[ 52 ], *cardPtr;
```

Имя структуры не является обязательным. Если определение структуры не специфицирует имени, переменные данного структурного типа могут объявляться только между закрывающей скобкой определения структуры и завершающей точкой с запятой.

При создании структурного типа специфицируйте имя структуры. Имя структуры необходимо для объявления новых переменных данного типа позднее в

программе, для объявления параметров структурного типа и, если структура используется подобно классу C++, для спецификации имени конструктора и деструктора.

Единственными допустимыми встроенными операциями, которые могут производиться над структурами, являются присваивание объекта структурного типа другому объекту того же типа, взятие адреса (&) структурного объекта, обращение к элементам структурного объекта (аналогично обращению к элементам класса) и применение операции sizeof для определения размера структуры.

Элементы структуры не обязательно хранятся в последовательных байтах памяти. Иногда в структуре имеются «дырки», поскольку некоторые компиляторы сохраняют конкретные типы данных только на определенных границах памяти (границе половинного слова, слова, двойного слова)

1.1. Структура программы на языке C++

Сама по себе программа на языке C++ представляет собой текстовый файл, в котором представлены конструкции и операторы данного языка в заданном программистом порядке.

Для того чтобы итоговая исполняемая программа содержала все необходимые реализации функций, используется компоновщик объектных кодов. Компоновщик - это программа, которая объединяет в единый исполняемый файл объектные коды создаваемой программы, объектные коды реализаций библиотечных функций и стандартный код запуска для заданной операционной системы. В итоге и объектный файл, и исполняемый файл состоят из инструкций машинного кода. Однако объектный файл содержит только результат перевода на машинный язык текста программы, созданной программистом, а исполняемый файл - также и машинный код для используемых стандартных библиотечных подпрограмм и для кода запуска.

Графически описанные этапы создания программы представлены на рис. 1.1.

Здесь исполняемый файл имеет расширение *.exe и может быть запущен обычным образом из соответствующей операционной системы: MS- DOS или Windows.

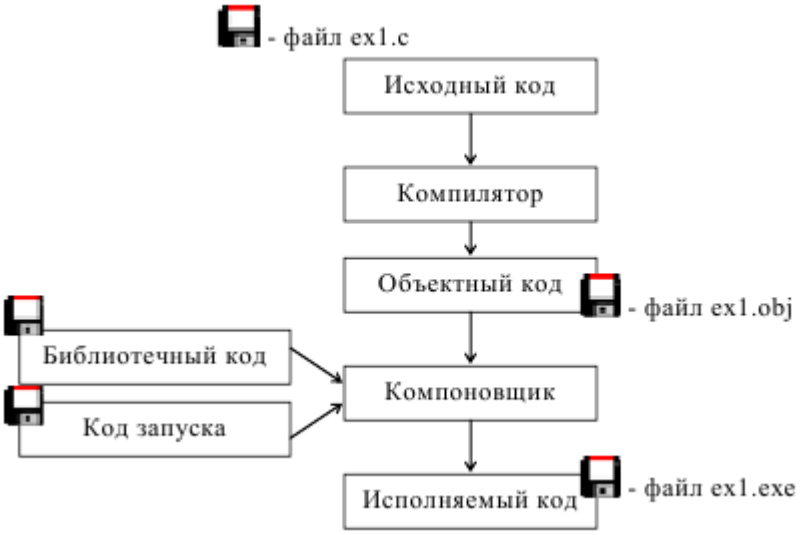


Рис. 1.1. Этапы создания программы

Рассмотрим более подробно пример программы листинга 1.1. Первая строка задает комментарии, т.е. замечания, помогающие лучше понять программу. Они предназначены только для чтения и игнорируются компилятором. Во второй строке записана директива #include, которая дает команду препроцессору языка C++ вставить содержимое файла ‘stdio.h’ на место этой строки при компиляции. В третьей строке определена функция с именем main, которая возвращает целое число (тип int) и не принимает никаких аргументов (тип void). Функция main() является обязательной функцией для всех программ на языке C++ и без ее наличия уже на этапе компиляции появляется сообщение об ошибке, указывающее на отсутствие данной функции. Обязательность данной функции обусловливается тем, что она является точкой входа в программу. В данном случае под точкой входа понимается функция, с которой начинается и которой заканчивается работа программы. Например, при запуске exe-файла происходит активизация функции main(), выполнение всех операторов, входящих в нее и завершение программы. Таким образом, логика всей программы заключена в этой функции. Фигурные скобки {} служат для определения начала и конца тела функции, т.е. в них содержатся все возможные операторы, которые описывают работу данной

функции. Следует отметить, что после каждого оператора в языке С++ ставится символ ‘;’. Таким образом, приведенный пример показывает общую структуру программ на языке С++.

1.2. Стандарты языка С++

В настоящее время имеется множество реализаций языка С++. В идеальном случае, написанная программа на одной реализации языка должна одинаковым образом выполняться и на любой другой реализации этого же языка. Для обеспечения этого условия существуют стандарты, описывающие основные конструкции С++ и правила их построения.

По мере того, как язык С постепенно развивался сообщество пользователей этого языка осознало, что нуждается в современном и строгом стандарте. В ответ на эти потребности Американский институт национальных стандартов (American National Standards Institute (ANSI)) в 1983 г. организовал комитет (X3J11) для разработки нового стандарта, который был принят в 1989 г. Этот стандарт (ANSI C) содержит определение как языка, так и стандартной библиотеки С. Затем международная организация по стандартизации (ISO) в 1990 г. приняла свой стандарт (ISO C), который по сути не отличается от стандарта ANSI C.

В 1994 г. возобновилась деятельность по разработке нового стандарта, в результате чего появился стандарт С99, который соответствует языку С++. Объединенный комитет ANSI/ISO развил исходные принципы предыдущего стандарта, являющийся основным на сегодняшний день.

1.3. Представление данных в языке С++

По существу программа есть не что иное, как обмен и преобразование разными типами данных. В связи с этим изучать программирование целесообразно со знакомства существующих типов данных. В табл. 1.1 представлены основные базовые типы данных языка С++.

Для того чтобы иметь возможность работать с тем или иным типом данных необходимо задать переменную соответствующего типа. Это осуществляется с использованием следующего синтаксиса:

<тип переменной> <имя_переменной>;

например, строка

int arg;

объявлет целочисленную переменную с именем arg.

Таблица 1.1. Основные базовые типы данных

Тип	Описание
int	Целочисленный тип 16 либо 32 бит
long int	Целочисленный тип 32 бит
short	Целочисленный тип 8 либо 16 бит
char	Символьный тип 8 бит
float	Вещественный тип 32 бит
double	Вещественный тип 64 бит

Отметим, что при выборе имени переменной целесообразно использовать осмысленные имена. При определении имени можно использовать как верхний, так и нижний регистры букв латинского алфавита. Причем первым символом обязательно должна быть буква или символ подчеркивания ‘_’.

Вот несколько примеров:

Правильные имена	Неправильные имена
arg	&arg
cnt	$cnt
bottom_x	bottom-x
Arg	2Arg
don t	don't

В приведенных примерах переменные arg и Arg считаются разными, т.к. язык C++ при объявлении переменных различает большой и малый регистры.

В отличии от многих языков программирования высокого уровня, в языке C++

переменные могут объявляться в любом месте текста программы.

1.4. Оператор присваивания

Для того чтобы задать то или иное значение переменной используется оператор присваивания, который записывается как знак ‘=’. Работу этого оператора представим на следующем примере:

int length = 5;

Здесь переменной с именем length присваивается значение 5, т.е. элемент слева от знака равенства является именем переменной, а элемент справа - это значение, присвоенное этой переменной. Сам символ ‘=’ называется оператором присваивания.

На первый взгляд, различие между именем переменной и значением переменной может показаться несущественным, однако рассмотрим такой пример обычного вычислительного оператора:

int i=2; i=i+1;

С математической точки зрения такая операция не имеет смысла, т.к. переменная i не может быть равна i+1. Однако данная запись в смысле операции присваивания вполне приемлема. Действительно, компьютер сначала определит значение переменной i, а затем прибавит 1 к этому значению и полученный результат присвоит снова переменной i. Таким образом, исходное значение переменной i увеличится на 1.

Так как оператор присваивания задает значение переменной, то оператор типа

2 0 = i;

не будет иметь смысла, поскольку число 20 будет интерпретироваться как константа, которой не может быть присвоено какое-либо другое значение.

Теперь рассмотрим такой пример. Допустим, что имеются две переменные разного типа:

short agr_short = -10; long arg_long;

и выполняется оператор присваивания

arg_long = arg_short;

В результате переменная arg_long будет иметь значение 10 и оператор присваивания выполнит автоматическое преобразование типов и потери данных не происходит. В другом случае, когда

float agr_f = 8.7; int arg_long; arg_long = arg_f;

в операторе присваивания произойдет потеря данных, т.к. целое число не может представлять числа, стоящие после запятой (точки). В результате переменная arg_long будет иметь значение 8. Для корректного преобразования
одного типа данных в другой используется операция приведения типов, имеющая следующий синтаксис:

<имя_переменной_1> = (тип_данных)<имя_переменной_2>;

Например,

arg_long = (long)arg_f;

1.5. Системы счисления

Любое число можно представлять в разной системе счисления. Во всех предыдущих примерах использовалось десятичная форма записи числа. Это значит, что число может быть разложено по степеням 10, например,

2145 = 2 • 103 +1-102 + 4-101 + 5-10°.

Однако компьютер оперирует двоичными числами, т.е. представляет число по степеням двойки, например:

64 = 1 • 26 + 0 • 25 + 0 • 24 + 0 • 23 + 0 • 22 + 0 • 21 + 0 • 20,

которое часто записывают в виде:

65	4	3	2	1	0
10	0	0	0	0	0

Учитывая, что в одном байте 8 бит, максимальное число, которое он может содержать равно

1 • 27 +1 • 26 +1 • 25 +1 • 24 +1 • 23 +122 +1 21 +1 20 = 255.

Таким образом, один байт информации может представлять десятичные числа в диапазоне от 0 до 255, т.е. 256 возможных значений.

Кроме десятичной и двоичной систем счисления при программировании часто используют шестнадцатеричную систему, т.е. числа с основанием 16. Для записи таких чисел недостаточно использовать цифры, поэтому для обозначения основания больше 9 добавляют буквы A - 10, B - 11, C - 12, D - 13, E - 14, F - 15 (или малого регистра a,b,c,d,e,f). Например, десятичное число 0 соответствует шестнадцатеричному 0, а десятичное 15, шестнадцатеричному F. Для представления числа 16 используется шестнадцатеричное 1461 + 0 • 160 = 10, а число 255 соответствует числу

15 •161 +15 •160 = F •161 + F •160 = FF.

Следует отметить, что каждая цифра шестнадцатеричного числа представляется четырьмя битами двоичного числа. Действительно

3 2 10

2 + 2 + 2 + 2 = 15 дает диапазон чисел от 0 до 15 или в шестнадцатеричной записи от 0 до F. Это свойство удобно для представления байтовых чисел, где каждая половинка байта представляется одним шестнадцатеричным числом.

Например,

FF = 11111111; OF = 00001111; 11 = 00010001 и т.д.

Для представления шестнадцатеричных чисел в языке C++ используется следующий синтаксис:

int var = 0xff; //число 255 int arg = 0xac; //число 172

1.6. Арифметические операции

В языке C++ довольно просто реализуются элементарные математические операции: сложения, вычитания, умножения и деления. Допустим, что в программе заданы две переменные

int a, b; с начальными значениями

тогда операции сложения, вычитания, умножения и деления будут выглядеть следующим образом:

представленные операции можно выполнять не только с переменными, но и с конкретными числами, например

c = 10+5; c = 8*4; float d; d = 7/2;

Результатом первых двух арифметических операций будут числа 15 и 32 соответственно, но при выполнении операции деления в переменную d будет записано число 3, а не 3,5. Это связано с тем, что число 7 в языке C++ будет интерпретироваться как целочисленная величина, которая не может содержать дробной части. Поэтому полученная дробная часть 0,5 будет отброшена. Для реализации корректного деления одного числа на другое следует использовать такую запись: d = 7.0/2;

a=4;

b=8;

int c;

c = a+b; c = a-b; c = a*b; c = a/b; //сложение двух переменных //вычитание //умножение //деление

или

d = (float)7/2;

В первом случае вещественное число делится на два и результат (вещественный) присваивается вещественной переменной d. Во втором варианте выполняется приведение типов: целое число 7 приводится к вещественному типу float, а затем делится на 2. Второй вариант удобен, когда выполняется деление одной целочисленной переменной на другую:

```
int a,b; a = 7; b = 2; d = a/b;
```

В результате значение d будет равно 3, но если записать

```
d = (float ) a/b;
```

то получим значение 3,5. Здесь следует также отметить, что если переменная d является целочисленной, то результат деления всегда будет записан с отброшенной дробной частью. Аналогичные правила справедливы для всех арифметических операций.

В заключении рассмотрения работы с арифметическими операциями отметим, что приоритет операций умножения и деления выше приоритета операций сложения и вычитания. Это означает, что сначала выполняются операции умножения и деления и только затем операции сложения и вычитания. Следующий пример демонстрирует приоритет арифметических операций:

```
double n=2, SCALE = 1.2;
double arg = 25.0 + 60.0*n/SCALE;
```

В приведенном примере сначала будет выполнена операция умножения, затем деления и, наконец, сложения. То есть порядок вычисления соответствует математическим правилам. Для того чтобы изменить порядок вычисления (поменять приоритеты) используются круглые скобки как показано ниже

```
double arg = (25.0 + 60.0)*n/SCALE;
```

Здесь сначала выполняется операция сложения и только затем операции умножения и деления.

Кроме рассмотренных арифметических операций в C++ имеется полезная операция деления по модулю. Ее результатом является остаток от деления одного целого числа на другое.

Так выражение

```
int a = 13 % 5;
```

означает, что число 13 делится по модулю 5. Учитывая, что число 5 дважды входит в число 13, то остаток получается равный 3. Эту операцию можно реализовать и на основе стандартных арифметических операций следующим образом:

```
int a = 13 - 13/5*5;
```

Следует отметить, что операция целочисленного деления % может быть реализована только для целых чисел и целочисленных переменных и не применима к другим типам данных.

Для простоты программирования в языке С++ реализованы компактные операторы инкремента и декремента, т.е. увеличения и уменьшения значения переменной на 1 соответственно. Данные операторы могут быть записаны в виде

```
i++;     // операция инкремента
++i;     // операция инкремента
i--;     // операция декремента
--i;     // операция декремента
```

Разницу между первой и второй формами записи данных операторов можно продемонстрировать на следующем примере:

```
int i=10,j=10;
int a = i++;    //значение a = 10; i = 11;
int b = + + j;  //значение b = 11; j = 11;
```

Из полученных результатов видно, что если оператор инкремента стоит после имени переменной, то сначала выполняется операция присваивания и только затем операция инкремента. Во втором случае наоборот, операция инкремента реализуется до присвоения результата другой переменной. Поэтому значение a = 10, а значение b = 11. В первом случае говорят о постпрефиксной форме, а во втором - о префиксной. Подобный приоритет операции инкремента остается справедливым и при использовании арифметических операций,

Например:

```
int a1=4, a2=4;
double b = 2.4*++a1;       //результат b =     12.0
double c = 2.4*a2++;       //результат c =     9.6
```

Из приведенного примера видно, что операция инкремента (декремента) обладает более высоким приоритетом, чем операция умножения (соответственно и деления). Для того чтобы изменить приоритеты используются круглые скобки.

Операция декремента действует аналогично операции инкремента с той лишь разницей, что она уменьшает значение переменной на 1.

1.7. Поразрядные операции языка C++

Поразрядные операции состоят из четырех основных операций: отрицание, логическое И, логическое ИЛИ и исключающее ИЛИ. Рассмотрим данные операции по порядку.

При выполнении операции поразрядного отрицания все биты, равные 1, устанавливаются равными 0, а все биты равные нулю, устанавливаются равными 1. Для выполнения данной операции в языке C++ используется символ как показано в следующем примере:

unsigned char var = 153; //двоичная запись 10011001 unsigned char not = ~var; //результат 01100110 (число 102)

В результате переменная not будет содержать число 102. В ходе выполнения операции поразрядного И результирующий бит будет равен 1, если оба бита в соответствующих операндах равны 1, т.е.

10010011 & 00111101 даст результат 00010001.

Для выполнения операции логического И используется символ & следующим образом:

unsigned char var = 153; //двоичная запись 10011001 unsigned char mask = 0x11;
// число 00010001 (число 17)

unsigned char res = var & mask; // результат 00010001

или

var &= mask; // то же самое, что и var = var & mask;

В ходе выполнения двоичной операции ИЛИ результирующий бит устанавливается равным 1, если хотя бы один бит соответствующих операндов равен 1. В противном случае, результирующее значение равно 0. Для выполнения данной логической операции используется символ ‘|’ как показано ниже:

unsigned char var = 153; //двоичная запись 10011001 unsigned char mask = 0x11;

// число 00010001

unsigned char res = var | mask; // результат 10011001

Также допускается применение такой записи

var |= mask; // то же самое, что и var = var | mask;

Наконец, при операции исключающее ИЛИ результирующий бит устанавливается равным 0, если оба бита соответствующих операндов равны 1, и 1 в противном случае. Для выполнения данной операции в языке C++ используется символ ‘A’:

unsigned char var = 153; //двоичная запись 10011001

unsigned char mask = 0x11; // число 00010001

unsigned char res = var л mask; // результат 10001000

или

var Л= mask; // то же самое, что и var = varл mask;

Рассмотрим примеры использования логических операций, которые часто применяются на практике. Самой распространенной по использованию является операция логического И. Данная операция обычно используется совместно с так называемыми масками. Под маской понимают битовый шаблон, который служит для выделения тех или иных битов числа, к которому она применяется. Например, если требуется определить, является ли нулевой бит числа установленным в 1 или нет, то для этого задается маска 00000001, которая соответствует числу 1 и выполняется операция поразрядного И:

unsigned char flags = 3; // 00000011 unsigned char mask = 1; // 00000001

if((flag & mask) == 1) printf("Нулевой бит включен"); else printf("Нулевой бит выключен");

Здесь переменная flags, представленная одним байтом, содержит восемь флаговых битов. Для того чтобы узнать установлен или нет нулевой флаговый бит задается маска со значением 1 и выполняется операция логического И. В результате все биты переменной flags будут равны нулю за исключением нулевого, если он изначально имел значение 1. Таким образом, маска является шаблоном, который как бы накладывается на битовое представление числа, из которого

выделяются биты, соответствующие единичным значениям маски. Рассмотренный пример показывает, как одна байтовая переменная flags может содержать восемь флаговых значений и тем самым экономить память ЭВМ.

Следующим примером использования логических операция является возможность включать нужные биты в переменной, оставляя другие без изменений. Для этого используется логическая операция ИЛИ. Допустим, в переменной flags необходимо установить второй бит равным 1. Для этого задается маска в виде переменной mask = 2 (00000010) и реализуется операция логического ИЛИ:

unsigned char flags = 0; // 00000000 unsigned char mask = 2; // 00000010 flags |= mask;

Этот код гарантирует, что второй бит переменной flags будет равен 1 без изменений значений других битов.

Для отключения определенных битов целесообразно использовать две логические операции: логическое И и логическое НЕ. Допустим, требуется отключить второй бит переменной flags. Тогда предыдущий пример запишется следующим образом:

unsigned char flags = 0; // 00000000 unsigned char mask = 2; // 00000010 flags = flags & ~mask;

или

flags &= ~mask;

Работа этих двух операций заключается в следующем. Приоритет операции НЕ выше приоритета операции И, поэтому переменная mask в двоичной записи будет иметь вид 11111101. Применяя операцию логического умножения переменной flags с полученным числом ~mask все биты останутся неизменными, кроме второго, значение которого будет равно нулю.

Наконец, операция исключающее ИЛИ позволяет переключать заданные биты переменных. Идея переключения битов основывается на свойствах операции исключающего ИЛИ: 1A1 = 0, 1A0 = 1, 0A0 = 0 и 0A1 = 1. Анализ данных свойств

показывает, что если значение бита маски будет равно 1, то соответствующий бит переменной, к которой она применяется, будет переключен, а если значение бита маски равно 0, то значение бита переменной останется неизменным. Следующий пример демонстрирует работу переключения битов переменной flags.

unsigned char flags = 0; //00000000 unsigned char mask = 2; //00000010 flags л= mask; //00000010

flags л= mask; //00000000

Кроме логических операций в языке C++ существуют операции поразрядного смещения битов переменной. Операция смещения битов влево определяется знаком << и смещает биты значения левого операнда на шаг, определенный правым операндом, например, в результате выполнения команды

10001010 << 2;

получится результат 00101000. Здесь каждый бит перемещается влево на две позиции, а появляющиеся новые биты устанавливаются нулевыми. Рассмотрим особенности действия данной операции на следующем примере:

int var = 1;

var = var <<1; //00000010 - значение 2

var <<= 1; //00000100 - значение 4

Можно заметить, что смещение битов переменной на одну позицию влево приводит к операции умножения числа на 2. В общем случае, если выполнить сдвиг битов на n шагов, то получим результат равный умножению переменной на 2n . Данная операция умножения на число 2n является более быстрой, чем обычное умножения, рассматриваемое ранее.

Аналогично, при операции смещения вправо >> происходит сдвиг битов переменной на шаг, указанный в правом операнде. Например, сдвиг

00101011 >> 2;

приведет к результату 00001010. Здесь, также как и при сдвиге влево, новые появляющиеся биты устанавливаются равными нулю. В результате выполнения последовательностей операций

int var = 128; //1000000

var = var >> 1; //0100000 - значение 64

var >>= 1; //0010000 - значение 32

значение переменной var каждый раз делится на 2. Поэтому сдвиг var >>= n можно использовать для выполнения операции деления значения переменной на величину 2n.

1.8. Директивы препроцессора

Почти все программы на языке C++ используют специальные команды для компилятора, которые называются директивами. В общем случае директива - это указание компилятору языка C++ выполнить то или иное действие в момент компиляции программы. Существует строго определенный набор возможных директив, который включает в себя следующие определения:

#define, #elif, #else, #endif, #if, #ifdef, #ifndef, #include, #undef.

Директива #define используется для задания констант, ключевых слов, операторов и выражений, используемых в программе. Общий синтаксис данной директивы имеет следующий вид:

#define <идентификатор> <текст>

или

#define <идентификатор> (<список параметров>) <текст>

Следует заметить, что символ после директив не ставится. Приведем примеры вариантов использования директивы #define.

Листинг 1.2. Примеры использования директивы #define.

```
#include <stdio.h>
#define TWO 2 #define FOUR TWO*TWO
#define PX printf("X равно %d.\n", x)
#define FMT «X равно %d.\n»
#define SQUARE(X) X*X int main()
{
int x = TWO;
```

```
PX;
x = FOUR;
printf(FMT, x);
x = SQUARE(3);
PX;
return 0;
}
```

После выполнения этой программы на экране монитора появится три строки:

X равно 2.

X равно 4.

X равно 9.

Директива #undef отменяет определение, введенное ранее директивой #define. Предположим, что на каком-либо участке программы нужно отменить определение константы FOUR. Это достигается следующей командой:

```
#undef FOUR
```

Интересной особенностью данной директивы является возможность переопределения значения ранее введенной константы. Действительно, повторное использование директивы #define для ранее введенной константы FOUR невозможно, т.к. это приведет к сообщению об ошибке в момент компиляции программы. Но если отменить определение константы FOUR с помощью директивы #undef, то появляется возможность повторного использования директивы #define для константы FOUR.

Для того чтобы иметь возможность выполнять условную компиляцию, используется группа директив #if, #ifdef, #ifndef, #elif, #else и #endif.

Приведенная ниже программа выполняет подключение библиотек в зависимости от установленных констант.

```
#if defined(GRAPH)
#include <graphics.h>
#elif defined(TEXT)
#include <conio.h>
```

#else

#include <io.h>

#endif

Данная программа работает следующим образом. Если ранее была задана константа с именем GRAPH через директиву #define, то будет подключена графическая библиотека с помощью директивы #include. Если идентификатор GRAPH не определен, но имеется определение TEXT, то будет использоваться библиотека текстового ввода/вывода. Иначе, при отсутствии каких-либо определений, подключается библиотека ввода/вывода. Вместо словосочетания #if defined часто используют сокращенные обозначения #ifdef и #ifndef и выше приведенную программу можно переписать в виде:

```
#ifdef GRAPH
#include <graphics.h> //подключение графической библиотеки #ifdef TEXT
#include <conio.h> //подключение текстовой библиотеки #else
#include <io.h>       //подключение библиотеки ввода-вывода
#endif
```

Отличие директивы #if от директив #ifdef и #ifndef заключается в возможности проверки более разнообразных условий, а не только существует или нет какие-либо константы. Например, с помощью директивы #if можно проводить такую проверку:

```
#if SIZE == 1
#include <math.h> // подключение математической библиотеки
#elif SIZE > 1
#include <array.h> // подключение библиотеки обработки массивов
#endif
```

В приведенном примере подключается либо математическая библиотека, либо библиотека обработки массивов, в зависимости от значения константы SIZE.

Данные директивы иногда используются для выделения нужных блоков программы, которые требуется использовать в той или иной программной

//подключение графической библиотеки //подключение текстовой библиотеки

//подключение библиотеки ввода-вывода реализации. Следующий пример демонстрирует работу такого программного кода.

Листинг 1.3. Пример компиляции отдельных блоков программы.

```
#include <stdio.h>
#define SQUARE
int main()
{
int s = 0;
int length = 10;
int width = 5;
#ifdef SQUARE
s=length*width;
#else
s=2*(length+width);
#endif return 0;
}
```

В данном примере происходит вычисление либо площади прямоугольника, либо его периметра, в зависимости от того определено или нет значение SQUARE. По умолчанию программа вычисляет площадь прямоугольника, но если убрать строку #define SQUARE, то программа станет вычислять его периметр.

Используемая в приведенных примерах директива #include позволяет добавлять в программу ранее написанные программы и сохраненные в виде файлов. Например, строка

#include <stdio.h>

указывает препроцессору добавить содержимое файла stdio.h вместо приведенной строки. Это дает большую гибкость, легкость программирования и наглядность создаваемого текста программы. Есть две разновидности директивы #include:

#include <stdio.h> - имя файла в угловых скобках

и

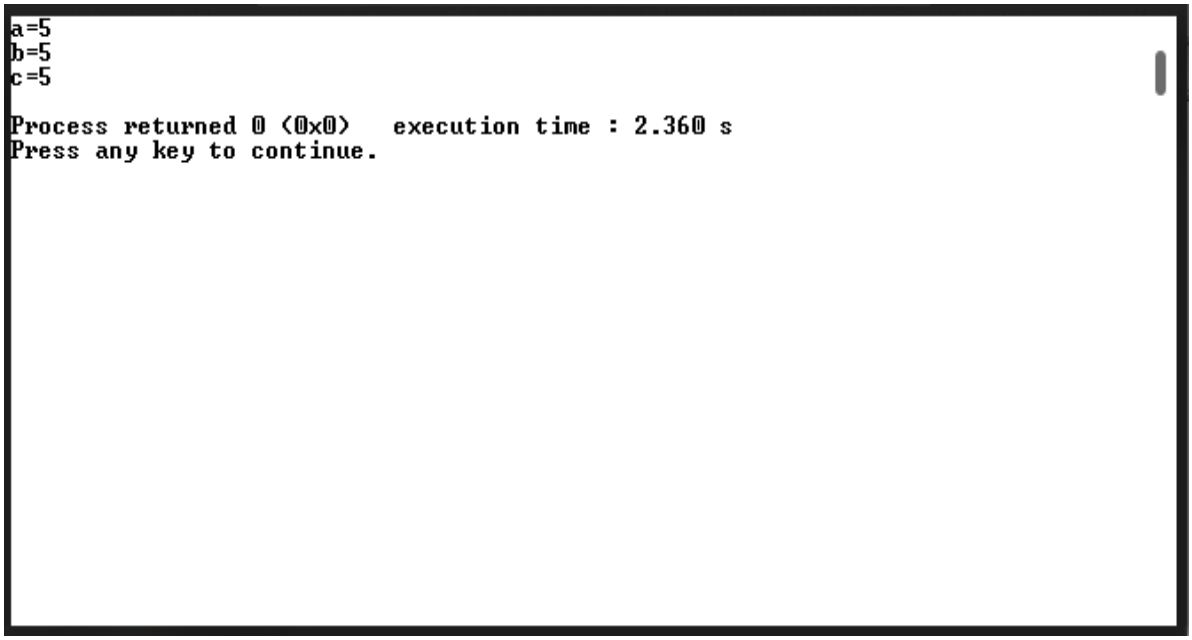
#include «mylib.h» - имя файла в кавычках

Угловые скобки сообщают препроцессору о том, что необходимо искать файл (в данном случае stdio.h) в одном или нескольких стандартных системных каталогах. Кавычки свидетельствуют о том, что препроцессору необходимо сначала выполнить поиск файла в текущем каталоге, т.е. в том, где находится файл создаваемой программы, а уже затем - искать в стандартных каталогах.

Практическая часть

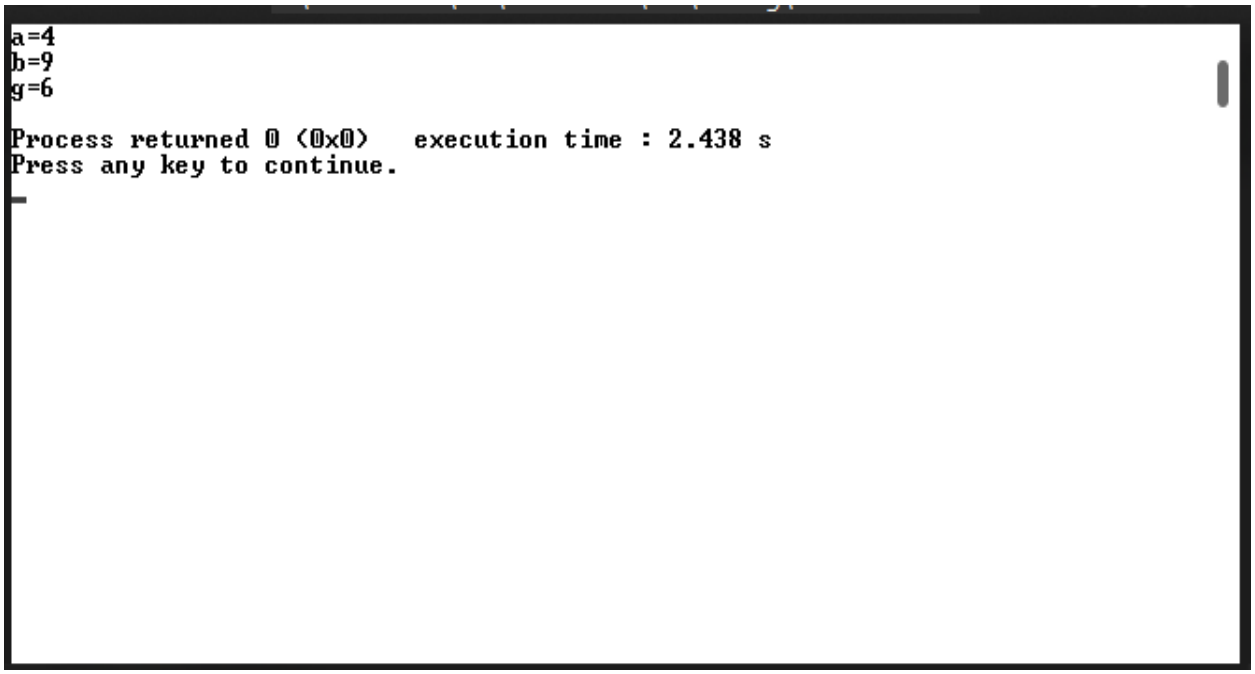
1) Даны два числа a и b. Найти их среднее арифметическое: (a + b) /2:

```
#include  < iostream >
#include  < cmath >
using namespace std ;

int main ( )
{
   float a , b , c ;
   cout << " a = " ;  cin >> a ;
   cout << " b = " ;  cin >>b ;
   c = ( a + b ) / 2 ;
   cout << " c = " << c << endl ;
   return 0 ;
}
```



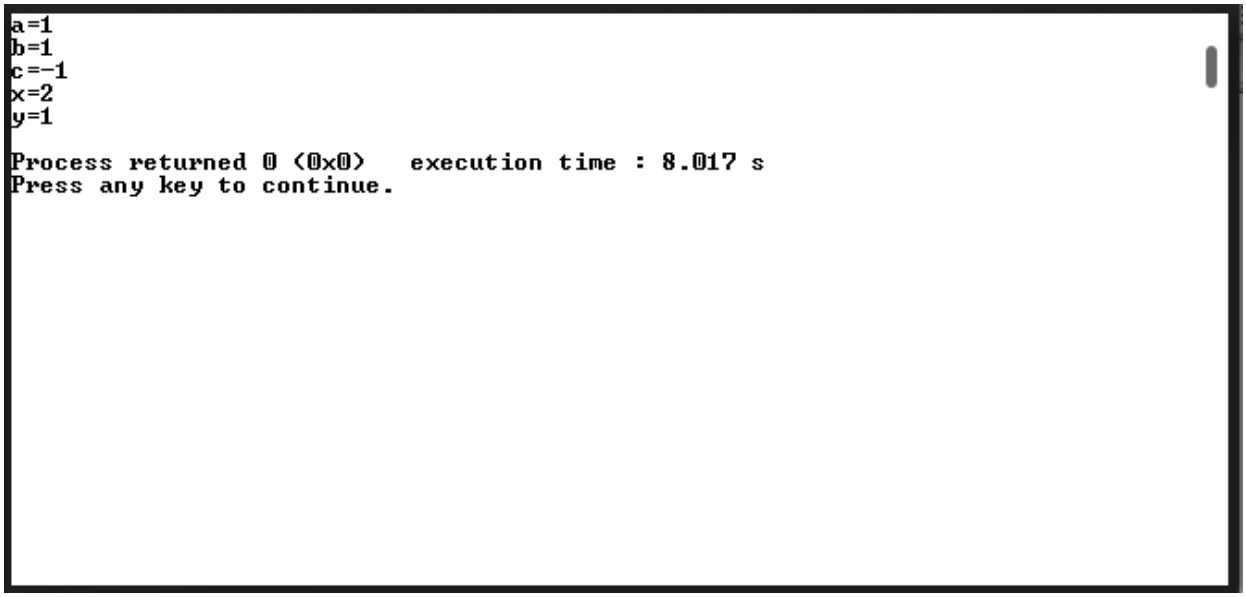
2) Даны два неотрицательных числа a и b. Найти их среднее геометрическое:

```
#include  < iostream >
#include  < cmath >
using namespace std ;

int main ( )
{
   float a , b , g ;
   cout << " a = " ; cin >> a ;
   cout << " b = " ;  cin >> b ;
   g = sqrt ( a * b ) ;
   cout << " g = " << g << endl ;
   return 0 ;
}
```



3) Даны три целых числа. Найти количество положительных и количество отрицательных чисел в исходном наборе:

```
#include  < iostream >
using namespace std ;
int main ( )
{
   int a , b , c ;
   int  x ; x = 0 ;
   int  y ; y = 0 ;
   cout << " a = " ; cin >> a ;
   cout << " b = " ;   cin >> b ;
   cout << " c = " ; cin >> c ;
   if ( a > 0 )
   x ++ ;
   if ( b > 0 )
   x ++ ;
   if ( c > 0 )
   x ++ ;
   cout << " x = " << x << endl ;
   if ( a < 0 )
   y ++ ;
   if ( b < 0 )
   y ++ ;
   if ( c < 0 )
   y ++ ;
   cout << " y = " << y << endl ;
   return 0 ;
}
```



Заключение

При разработке большинства языков программирования преследуются сугубо прикладные цели. Например, при создании языка программирования Pascal преследовалась цель преподавание основ программирования. При проектировании BASIC планировалась высокая степень сходства с английским языком, что обеспечивало простоту изучения этого языка людьми, ранее не имевшими дело с компьютерами. При разработке языка C++ преследовалась цель создания инструментального средства, предназначенного для создания программ разной степени сложности. Язык C++ ориентирован на удовлетворение потребностей программистов. Он предоставляет доступ к аппаратным средствам и позволяет оперировать отдельными битами оперативной памяти. Он включает широкий набор операторов, позволяющих программисту выражать свои идеи в компактном виде. Язык программирования C++ менее строгий, чем, например, язык Pascal в смысле ограничений свободы действий программиста. С одной стороны, эта гибкость является достоинством, но с другой – таит в себе некоторую опасность. Достоинство заключается в том, что многие задачи, например, преобразование типов переменных, в языке C++ решаются достаточно просто. Вместе с тем эта свобода может приводить к ошибкам, которые не возможны в других языках программирования. Таким образом, язык C++ предоставляет большую свободу действий, но и накладывает на программиста большую степень ответственности. Язык программирования C++ имеет свои недостатки. Самым большим из них считается использование указателей. При этом программист может совершать такие программные ошибки, которые, затем, довольно трудно обнаружить. Также компактность языка C++ в сочетании с большим количеством операторов дает возможность создавать код, понимание которого чрезвычайно затруднительно. Несмотря на указанные недостатки, данный язык программирования является лидером по использованию при написании приложений разной степени сложности.

Список использованной литературы:

1. С.М. Наместников «Основы программирования на С++» Ульяновск 2007
2. Н. А. Аленский «Основы программирования на языке С++» Минск 2005
3. Харви Дейтел Пол Дейтел «Как программировать на С++»
4. www.help-s.ru