ГОСУДАРСТВЕННЫЙ КОМИТЕТ СВЯЗИ, ИНФОРМАТИЗАЦИИ И ТЕЛЕКОММУНИКАЦИОННЫХ ТЕХНОЛОГИИ РЕСПУБЛИКИ УЗБЕКИСТАН

НУКУССКИЙ ФИЛИАЛ

ТАШКЕНТСКОГО УНИВЕРСИТЕТА

ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

Факультет _____*«Компьютер инжиниринга»*_____

Направление_____*Телекоммуникационные технологии*_____

КУРСОВАЯ

РАБОТА

По предмету _____*« Программирования на языке С++»*_____

На тему: _____« Переменные на языке С++»_____

Выполнил(а): _*студент 2^{В} курса_Лазарев А.*_

Принял(а): ____*Ядгаров Ш.*___

Нукус 2014 г.

Курсовая работа

По дисциплине «Программирование на С++»

На тему «Переменные на языке С++»

Содержание

Введение

До сих пор мы рассматривали переменные, которые имели только одно значение, которые могли содержать в себе только одну величину определенного типа. Исключением являлись лишь строковые переменные, которые представляют собой совокупность данных символьного типа, но и при этом мы говорили о строке, как об отдельной величине.

Вы знаете, что компьютер предназначен в основном для облегчения работы человека с большими информационными объемами. Поэтому во всех существующих языках имеются типы переменных, отвечающие за хранение больших массивов данных.

Вскоре мы обсудим типы данных **float** (для спецификаций действительных чисел , то есть чисель с десятичными запятыми типа 3.4, 0.0, -11.19) и **char** (для спецификации символьных данных; переменная **char** может хранить только одну строчную букву или одну прописную букву, одну цифру, один специальный символ типа *, $ и т. д.).

Имя переменной – это любой допстимый идентификатор. Идентификатором называется последовательность символов, содержающая буквы, цифры и символы подчеркивания (_), которая не начинается с цифры. В C++ допускаются идентификаторы любой длины, но ваша система или среда C++ могут налагать некоторые ограничения на длину идентификаторов. Язык C++ чувствителен к регистру – прописные и строчные буквы различаются , так что a1 и A1 — это разные идентификаторы.

Объявления могут размещаться в программе почти всюду. Однако, объявления переменных должны предшествовать их использованию в программе. Например, в программе вместо использования одного объявления для всех трех переменных можно было использовать три отдельных объявления.

1.Переменные

Переменная — это место в памяти компьютера, где может - сохраняться некоторое значение для использования его в программе. Данное объявление определяет, что переменные **a1 , b2** и **summa** имеют тип данных int; это значит, что эти переменные всегда будут содержать целые значения, т.е. целые числа, такие как 7, -11, 0, 31914. Все переменные в программе должны объявляться с указанием имени и типа данных, прежде чем они могут быть использованый в программе. Несколько переменных одного типа могут быть объявлены в одном или в нескольких объявлениях. Мы могли бы написать три объявления, по одному для каждой переменной , но предыдущее объявление более компактно.

1.2.Тип, имя и значение переменной

В этом параграфе даются основные понятия о переменных, их именах, значениях, данных и типах.

Понятие переменной

Переменные задаются именами, определяющими области памяти, в которых хранятся значения переменных. Значениями переменных могут быть данные различных типов (целые, вещественные числа, последовательности символов и так далее).

Переменная в программе представлена именем и служит для обращения к данным определенного типа. Конкретное значение переменной хранится в ячейках оперативной памяти.

Тип переменной

Тип переменной определяется типом данных, которые могут быть значениями переменной. Значениями переменных числовых типов являются числа, логических – True или False, строковых последовательности символов и так далее.

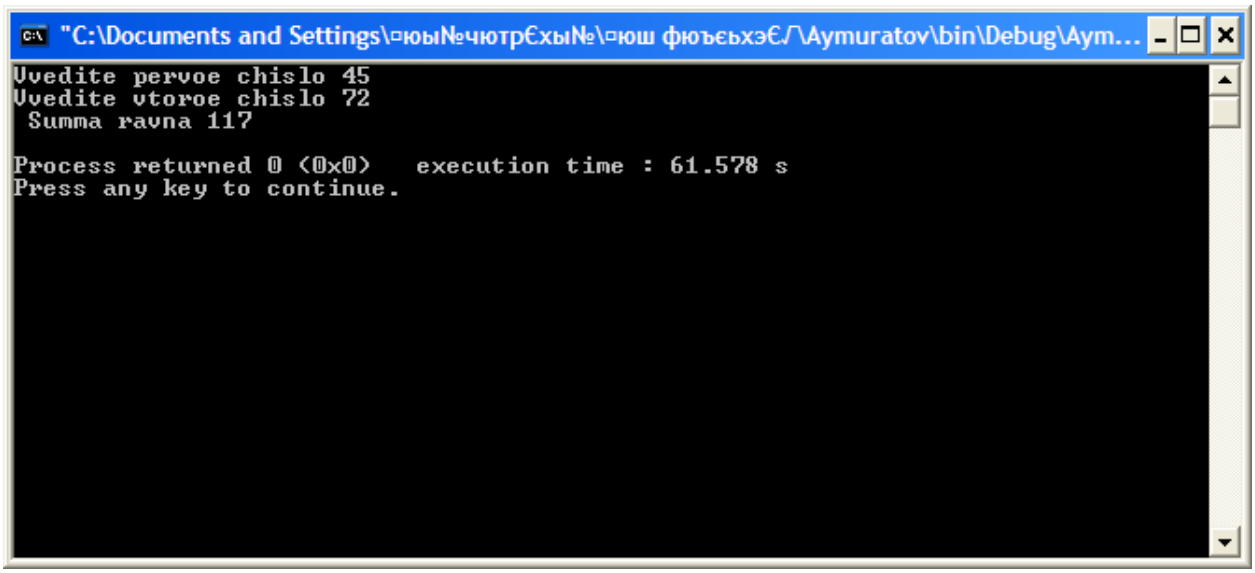
Над различными типами данных допустимы различные операции. Над числовыми арифметические операции, над логическими – логические операции, над строковыми – операции преобразования символтных строк и так далее.

Пример

```
#include <iostream>


using namespace std;


int main()
{
   int a,b,c;
   cout <<"Vvedite pervoe chislo ";
   cin >>a;
   cout <<"Vvedite vtoroe chislo ";
   cin >>b;
   c = a+b;
   cout <<" Summa ravna "<< c << endl;
   return 0;
}
```



Объявление

int a1; // первое из складываемых чисел

можно было бы поместить непосредственно перед строкой

cin » a1; // прочитать первое число в numberl

Объявление

int b2; // второе из складываемых чисел

можно было бы поместить перед строкой

cin » b2; // прочитать второе число в number2

и объявление

int summa; // сумма numberl и number2

можно было поместить перед строкой

summa = a1 + b2; // сложить числа; Записать сумму в summa.

Оператор

Cout << “Введите первое целое число” // приглашение

Печатает на экране буквенное сообшение **Введите первое целое число** и позиционирует курсор на начало следующей строки. Это сообщение называется *приглащением,* потому что оно предлагает пользователю выполнить некоторое действие. О предыдущем операторе можно сказать так: **<<cout** *получает* символьную строку “**Введите первое число”>>**

Оператор

cin >> a1; // чтение целого

Использует объект входного потока **cin** и операцию взять из потока >>, чтобы получить от пользователья значение. Объект **cin** забирает вводимую информацию из стандартного потока ввода, которым обычно является клавиатура. О предыдущем операторе можно сказать так: **<<cin** дает значение первого целого числа>>.

Когда компьютер выполняет предыдущий оператор, он ждет от пользователя ввода значения переменной **a1**. В ответ пользователь набирает на клавиатуре целое число (в символьном представлении) и затем нажимает

клавишу возврата — **return** (называемую иногда клавишей ввода — **Enter**), чтобы послать это число в компьютер. Компьютер затем присваевает это число, или значение, переменной **a1.** Любое последующее обращение в программе к **a1** будет использовать это самое значение.

Объекты потоков **cout** и **cin** вызывают взаимодействие между пользователем и компьютером. Поскольку это взаимодействие напоминает диалог, часто говорят о *диалоговом* расчете или *интерактивном* расчете.

Оператор

Cout << " Введите второе целое число\n"; //приглашение

Печатает на экране сообщение **Введите второе целое число** и затем позиционирует курсор на начало следующей строки. Этот оператор приглашает пользователя выполнить действие. Оператор

cin >> b2; //чтение целого

получает от пользователя значение переменной **b.**

Оператор присваивания

Sum = a1 + b2; //присваивание значения сумме

Рассчитывает сумму переменных **a1** и **b2** и присваивает результат переменной **sum,** используя операциюприсваивания =. Оператор читается так: **<< sum** получает значение, равное **a1 + b2>>.** Оператор присваивания используется в большинстве расчетов. Операция = и операция + называются бинарными операциями, потому что каждая из них имеет по два операнда. В случае операции + этими операндами являются **a1 + b2.** В случае операции = двумя операндами являются **sum** и значение выражения **a1 + b2.**

Оператор

Cout << " Сумма " << summa << endl; //печать суммы

Печатает символьную строку **Сума равна** , затем численное значение переменной **summa,** за которым следует **endl** (аббревиатура словосочетания <<endl line>> - конец строки) так называемый манипулятор потока. Манипулятор **endl** выводит символ новой строки и затем «очищает буфер вывода ». Это просто означает, что в некоторых системах, где выводы накопляются в вычислительной машине до тех пор, пока их не станет достаточно, чтобы «имело смысл печатать на экране», **endl** вызывает

немедленную печать на экране всего накопленного. Заметим, что предыдущий оператор выводит множество значений различных типов. Операция передачи в поток «знает», как выводить каждый из типов данных. Многократное использование операции поместить в поток (<<) в одном операторе называется сцепленной операцией поместить в поток. Таким образом , не обязательно иметь множество операций вывода для вывода множества фрагментов данных. В операторах вывода можно также выполнить вычисления. Мы могли бы объединить два предыдущих оператора в один

cout « "Сумма равна " «a1 + b2 « endl;

Правая фигура скобка } информирует компьютер о том, что функция **main** окончена.

Мощным свойством C++ является предоставление пользователям возможности создавать свои собственные типы данных.

1.3.Концепции памяти переменных

Имена переменных, такие как **al, b2** и **summa,** в действительности соответствуют областям в памяти компьютера. Каждая переменная имеет *имя, тип, размер* и *значение*.

В программе сложения на рис. 1.6 при выполнении оператора

cin » a; // прочитать первое число

значение, введенное пользователем, помещается в область памяти, которой компилятор присвоил имя **a1**. Допустим, пользователь вводит число 45 как значение **a1.** Компилятор разместит 45 в областей памяти **a1,** как показано на рис. 1.7

Всякий раз, когда значение помещается в ячейку памяти, оно замещает предыдущее значение, хранившееся в этом месте. Предыдущее значение при этом пропадает.
Возвращаясь к нашей программе сложения, допустим, что при выполнении оператора

cin » b2; // прочитать второе число

пользователь вводит значение 72. Это значение помещается в область памяти **b2** и теперь распределение памяти выглядит так, как показано на рис. 1.8. Заметим, что соседство этих областей в памяте необязательно.

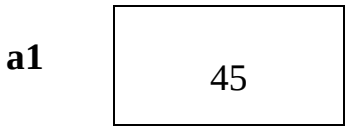


Рис. 1.7. Распределение памяти с указанием имени и значения переменной

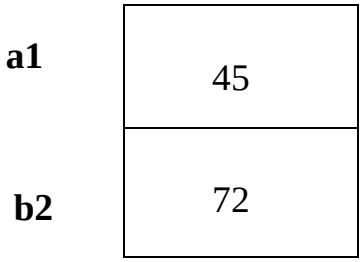


Рис. 1.8. Распределение памяти после ввода значений переменных

Когда программа получила значения **al** и **b2,** она складывает их и помещает сумму в переменную **summa**. Оператор

summa = al + b2; // сложить числа; записать сумму в summa

выполняет сложение и подразумевает уничтожение предыдущего значение, **summa.** Это происходит, когда вычисленное значение суммы **al** и **b2** помещается в область памяти **sum** (независимо от того, каково было прежнее значение **summa)**. После вычисления **summa** распределение памяти выглядит так, как показано на рис. 1.9. Заметим , что значения **al** и **b2** выглядят точно так же, как и до их исрользования при вычислении **summa**. Эти значения использовались, но не провали во время выполняемых компьютером вычислений. Таким образом, считывание значений из памяти - процесс неразрушающий.

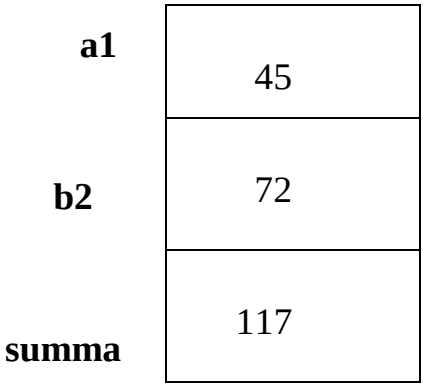


Рис. 1.9. Распределение памяти после вычислений

2. Переменные в массиве

Само массив — это последовательная группа ячеек памяти, имеющих одинаковое имя и одинаковый тип.

Массив в C++ - это последовательность элементов одного и того же типа.

Доступ к каждому элементу осуществляется по его положению в массиве. Эта форма называется индексацией.

2.1 Производные типы данных

Производные типы

* - указатель;

& - ссылка;

[ ] - массив;

( ) – функция;

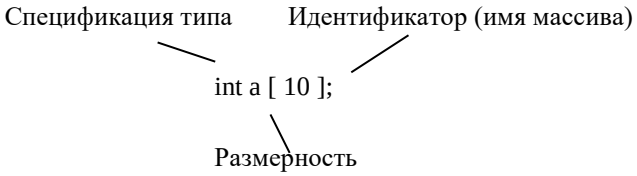
получаются из стандартных типов или классов с помощью операций объявления (их еще называют модификаторами типа).

Массив — это последовательная группа ячеек памяти, имеющих одинаковое имя и одинаковый тип.

Массив в C++ - это последовательность элементов одного и того же типа.

Доступ к каждому элементу осуществляется по его положению в массиве. Эта форма называется индексацией.

Описание массива состоит из спецификации типа, идентификатора и размерности:



Размерность массива должна быть больше или равна 1.

В программе можно обращаться как к массиву целиком (по его имени), так и к отдельным компонентам (элементам) массива. Например,

```
int i = 123;
a[2] = i;
i = a[7]
```

Значение размерности должно быть константным выражением, т.е. она должна быть известна на этапе трансляции. Это означает, что переменная не может использоваться (как в Паскале) для задания размерности массива.

2.2 Массивы. Синтаксис объявления

Массивы представляют собой производные типы (указатели также относятся к производным типам).

Объект типа "массив элементов заданного типа" представляет последовательность объектов этого самого типа, объединенных одним общим именем. Количество элементов массива является важной характеристикой самого массива, но не самого типа. Эта характеристика называется размерностью массива.

Приведем примеры объявления и определения массивов.

```
extern int intArray_1[];
```

Объявлен (именно объявлен - об этом говорит спецификатор extern) массив типа int, имя массива - intArray_1, разделители [] указывают на то, что перед нами объявление массива.

```
int intArray_2[10];
```

А это уже определение массива. Все тот же тип int, имя массива - intArray, между разделителями [и] находится константное выражение, значение которого определяет размерность массива.

Требование синтаксиса по поводу константного выражения между разделителями в определении массива может быть объяснено лишь тем, что информация о количестве элементов массива требуется до момента начала выполнения программы.

```
int intArray_3[] = {1,2,3}; // Это также определение массива.
```

Количество элементов массива становится известным транслятору при анализе инициализатора. Элементам массива присваиваются соответствующие значения из списка инициализаторов.

Еще одна форма определения массива:

```
int intArray_4[3] = {1,2,3};
```

В этом определении массива важно, чтобы количество элементов в инициализаторе массива не превышало значение константного выражения в описателе массива.

В результате выполнения этого оператора в памяти выделяется область, достаточная для размещения трех объектов-представителей типа int. Участку присваивается имя intArray_4. Элементы инициализируются значениями, входящими в состав инициализатора.

Возможна частичная инициализация массива. При этом значения получают первые элементы массива:

```
int intArray_5[3] = {1,2};
```

В этом определении массива означены лишь первые два элемента массива. Значение последнего элемента массива в общем случае не определено.

Здесь нужно отметить одну интересную особенность синтаксиса инициализатора массива. Речь идет о необязательной запятой в конце списка инициализаторов. По-видимому, е╤ назначение заключается в том, чтобы указывать на факт частичной инициализации массива.

Действительно, последний вариант (частично) инициализирующего оператора определения массива выглядит нагляднее:

```
int intArray_5[3] = {1,2,};
```

Последняя запятая предупреждает о факте частичной инициализации массива. Затраты на связывание запятой в конце списка инициализаторов со строго определенным контекстом частичной инициализации оказываются столь значительными, что последняя запятая традиционно (по крайней мере со времени выхода "Справочного руководства по языку программирования C++") оказывается всего лишь необязательным элементом любой (в том числе и полной) инициализации.

```
int intArray_6[3] = {1,2,3};
int intArray_6[3] = {1,2,3,};// Полная инициализация с запятой
int intArray_6[] = {1,2,3};
```

```
int intArray_6[] = {1,2,3,};
```

Между этими операторами не существует никакой разницы.
А вот в таком контексте

```
int intArray_6[3] = {1,2,}; // Частичная инициализация массива из трех элементов
```

Последняя запятая в фигурных скобках - не более как полезное украшение. Что-то недосказанное таится в таком операторе присвоения.

```
int intArray_7[];
```

А вот это некорректное объявление. Без спецификатора extern транслятор воспринимает это как ошибку. В скором времени мы обсудим причину этого явления.

2.3. Основные свойства массивов

Первое специфическое свойство массивов заключается в том, что определение массива предполагает обязательное указание его размеров. Зафиксировать размер массива можно различными способами (о них мы уже говорили), однако это необходимо сделать непосредственно в момент его объявления, в соответствующем операторе объявления.

В модулях многомодульной программы массив определяется в одном из модулей (в главном модуле) программы. В остальных модулях при объявлении этого массива используется спецификатор extern. Подобное объявление может быть включено и в главный модуль. Главное, чтобы транслятор мог различить объявления и собственно определение.

В объявлениях со спецификатором extern можно указывать произвольные размеры объявляемого массива (лишь бы они были описаны в виде константного выражения), а можно их и не указывать вовсе - транслятор все равно их не читает.

```
int intArray1[10] = {0,1,2,3,4,5,6,7,8,9};
extern intArray1[];
extern intArray1[1000];
```

Казалось бы, если транслятор все равно не читает значение константного выражения в объявлении, то почему бы там не записать выражение, содержащее переменные?

```
int ArrVal = 99;
extern intArray1[ArrVal + 1];
```

/*Однако этого сделать нельзя. ArrVal не константное выражение.*/

Но зато он очень строго следит за попытками повторной инициализации.

```
extern intArray1[10] = {9,9,9,};
```

/*Здесь будет зафиксирована ошибка. Хотя, если в объявлении не проверяется размерность массива, то какой смысл реагировать на инициализацию*/

Второе свойство массивов заключается в том, что объекту типа массив невозможно присвоить никакого другого значения, даже если это значение является массивом аналогичного типа и размерности:

```
char chArray_1[6];
char chArray_2[] = {'q', 'w', 'e', 'r', 't', 'y'};
Попытка использовать оператор присвоения вида
chArray_1 = chArray_2;
```

вызывает сообщение об ошибке, суть которой сводится к уведомлению, что выражение chArray_1 не является леводопустимым выражением.

4. Выражение и l-выражение

Доступ к объектам и функциям обеспечивается выражениями, которые в этом случае ссылаются на объекты.

Выражение, которое обеспечивает ссылку на константу, переменную или функцию, называется l-выражением. Имя объекта в C++ является частным случаем l-выражения.

В C++ допускается изменение значений переменных. Значения констант и функций в C++ изменению не подлежат. l-выражение называется модифицируемым l-выражением, либо леводопустимым выражением, если только оно не ссылается на функцию, массив или константу. Таким образом,

леводопустимыми выражениями называют l-выражения, которые ссылаются на переменные.

Следует заметить, что подобным образом ведет себя и константный указатель, с которым мы познакомились раньше. Он также требует немедленной инициализации (это его единственный шанс получить определенное значение) и не допускает последующего изменения собственного значения.

Часто указатель один "знает" место расположения участка памяти, выделенного операциями или функциями распределения памяти. Изменение значения этого указателя приводит к потере ссылки на расположенный в динамической памяти объект. Это означает, что соответствующая область памяти на все оставшееся время выполнения программы оказывается недоступной.

По аналогичной причине невозможна и операция присвоения, операндами которой являются имена массивов.

Операторы

```
intArray1 = intArray2;
intArray1[] = intArray2[];
```

не допускаются транслятором исключительно по той причине, что имя массива аналогично константному указателю. Оно является неизменяемым l-выражением, следовательно, не является леводопустимым выражением и не может располагаться слева от операции присвоения.

Массив символов (типа char) можно инициализировать и списком из отдельных символьных констант, так и строкой символов. При этом в случае строки символов массив получит в конце и нулевой символ, ограничивающий строку, например:

```
char ca_1 [ ] = { ‘C’, ‘+’, ‘+’ } ;

char ca_2 [ ] = “C++” ;
```

Здесь ca_1 будет размерности 3, а ca_2 - размерности 4 и содержать { ‘C’, ‘+’, ‘+’, ‘\0’ }.

Один массив не может инициализироваться другим массивом и не может быть присвоен другому массиву, например:

```
int a1 [ ] = { 0, 1, 2 } ;
int a2 [ ] = a1 ; // ошибка!!!
```

```
. . .
int a3 [ 3 ] ;
a3 = a1 ; // ошибка!!!
```

Чтобы скопировать один массив в другой, необходимо скопировать каждый элемент по очереди, например:

```
const int a_size = 7 ;
int a1 [ ] = { 0, 1, 2, 3, 4, 5, 6 } ;
…..
int a2 [ a_size ] ;
for ( int i =0; i < a_size; i++)
 a2[ i ] = a1 [ i ] ;
```

В качестве индекса массива может использоваться любое выражение, которое приводит к целочисленному значению.

В C++ нет контроля выхода за границы массива. Вся ответственность лежит на программисте!

Заметим, что при создании в динамической памяти с помощью выражения размещения безымянных массивов объектов (при инициализации указателей на массивы) инициализаторы не допускаются. Инициализатор в выражении размещения может проинициализировать только один объект. И дело здесь не в особых свойствах выражения размещения, а в особенностях языка и самого процесса трансляции.

Рассмотрим процессы, происходящие при выполнении оператора определения массива. Они во многом аналогичны процессам, происходящим при определении константного указателя:

- по константному выражению в описателе или на основе информации в инициализаторе определяется размер необходимой области памяти. Здесь сразу уже необходима полная информация о размерности массива. Размер области памяти равняется произведению размера элемента массива на размерность массива,
- выделяется память,
- адрес выделенной области памяти присваивается объекту, который по своим характеристикам близок константному указателю (хотя это объект совершенно особого типа).

Теперь можно вспомнить объявление, которое было рассмотрено нами в одном из прошлых разделов. Объявление массива

```
int intArray_7[];
```

воспринимается транслятором как ошибочное объявление исключительно по причине функционального сходства между объявлением массива и объявлением константного указателя. Массив, как и константный указатель должен быть проинициализирован в момент объявления.

2.4 Массив констант

Как уже известно, имя массива является константным указателем. Именно поэтому и невозможно копирование массивов с помощью простого оператора присвоения. Константный указатель "охраняет" область памяти, выделенную для размещения данного массива. При этом значения элементов массива можно изменять в ходе выполнения программы. Защитить их от изменения можно с помощью дополнительного спецификатора типа const. При этом массив должен быть проинициализирован непосредственно в момент определения:

```
const int cIntArray[] = {0,1,2,3,4,5,6,7,8,9};
```

Это аналог константного указателя на массив констант. Попытки изменения значения элементов массива пресекаются на этапе компиляции.

```
cIntArray[5] = 111; // Ошибка.
```

А вот от скрытого изменения значения элементы массива констант уберечь не удается.

```
const char cCH[] = "0123456789";
char CH[] = "0123456789";
CH[15] = 'X';

/* Выполнение этого оператора ведет к изменению строки cCH. */

cout << cCH << endl;
```

Список литературы

1.Как программировать на C++ (H. P. Datel)

2. Павловская_C++_ООП_Практикум

3. Павловская Т.А. C++. Программирование на языке высокого уровня

4. DirectX 9. Уроки программирования на C++