ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ С++

Распределение памяти. Указатели и ссылки

Никитин Михаил Евгеньевич, 2015

Указатели

- Указатели — один из самых важных и сложных аспектов C++.
- Благодаря указателям обеспечивается поддержка связных списков и динамического выделения памяти.
- Указатели позволяют функциям изменять содержимое своих аргументов

Указатели

- При рассмотрении темы указателей нам придется использовать такие понятия, как размер базовых C++-типов данных.
  - символы занимают в памяти один байт
  - целочисленные значения четыре
  - с плавающей точкой типа float четыре
  - с плавающей точкой типа double восемь

Указатели

- Указатели — это переменные, которые хранят адреса памяти.
- Чаще всего эти адреса обозначают местоположение в памяти других переменных.
- Например, если x содержит адрес переменной y, то о переменной, x говорят, что она "указывает" на y.

Указатели

- Переменные-указатели (или переменные типа указатель) должны быть соответственно объявлены. Формат объявления переменной-указателя таков:

тип *имя_переменной;

Указатели

- Чтобы объявить переменную p указателем на int-значение, используйте следующую инструкцию.

```
int *p;
```

- Для объявления указателя на float-значение используйте такую инструкцию.

```
float *p;
```

Указатели

- Рассмотрим еще один пример.

```
int *ip; // указатель на целочисленное значение
double *dp; // указатель на значение типа double
```

- Переменную ip можно использовать для указания на int-значения, а переменную dp на double-значения.
- Однако помните: не существует реального средства, которое могло бы помешать указателю ссылаться на "бог-знает-что". Вот потому-то указатели потенциально опасны.

Операторы, используемые с указателями

- С указателями используются два оператора: * и &
- Оператор & — унарный. Он возвращает адрес памяти, по которому расположен его операнд.
- Пример

```
int balance=4;
int *balptr = &balance;
```

в переменную balptr помещается адрес переменной balance.

Операторы, используемые с указателями

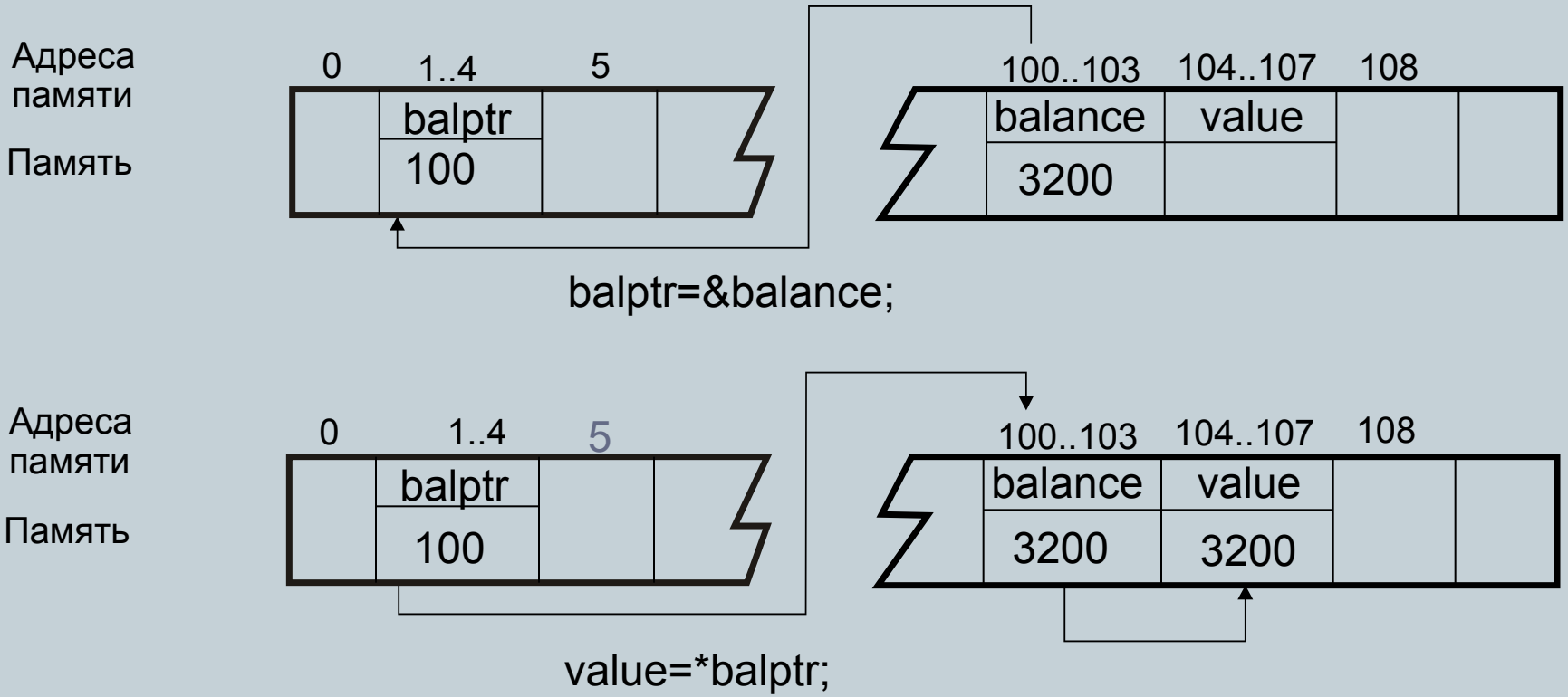
- Оператор работы с указателями * - это унарный оператор, но он обращается к значению переменной, расположенной по адресу, заданному его операндом.
- Другими словами, он ссылается на значение переменной, адресуемой заданным указателем.
- Пример

```
int* balptr = &a;
value = *balptr;
```

Операторы, используемые с указателями


```
#include <iostream>
using namespace std;
int main()
{
   int balance;
   int *balptr;
   int value;
   balance = 3200;
   balptr = &balance;
   value = *balptr;
   cout << "Баланс равен:" << value << '\n';
   return 0;
}
```

Операторы, используемые с указателями



Операторы, используемые с указателями

- Знак умножения (*) и оператор со значением "по адресу" (*) обозначаются одинаковыми символами «звездочка»
- Эти операции никак не связаны одна с другой.
- Имейте в виду, что операторы "*" и "&" имеют более высокий приоритет, чем арифметические операторы

О важности базового типа указателя


```
#include <iostream>
using namespace std;
int main()
{
    int balance;
    int *balptr;
    int value;
    balance = 3200;
    balptr = &balance;
    value = *balptr;
    cout << "Баланс равен:" << value << '\n';
    return 0;
}
```

Как C++-компилятор узнает, сколько необходимо скопировать байтов в переменную value из области памяти, адресуемой указателем balptr?

О важности базового типа указателя

- Ответ звучит так. Тип данных, адресуемый указателем, определяется базовым типом указателя.

```
int balance;
int *balptr;
int value;
```

- Переменные-указатели должны всегда указывать на соответствующий тип данных.

О важности базового типа указателя

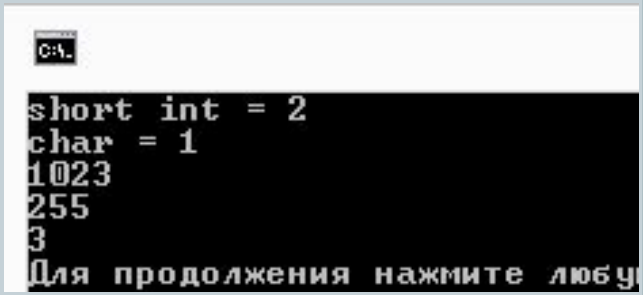
- Например, при объявлении указателя типа int компилятор "предполагает", что все значения, на которые ссылается этот указатель, имеют тип int.
- Корректен ли следующий кусок кода?.

```
int *p;
double f;
// ...
p = &f;
```

О важности базового типа указателя


```
#include <iostream>
using namespace std;
int main()
{
 short int a = 1023;
 cout << "short int = " << sizeof(short int) << endl;
 cout << "char = "<< sizeof(unsigned char) << endl;
 short int *ptr_a = &a;
 unsigned char *ptr_c = (unsigned char *)&a;
 cout << *ptr_a << endl;
 cout << (int)*ptr_c << endl;
 cout << (int)*(ptr_c + 1) << endl;
 return 0;
}
```

Что будет выведено на экран?



Присваивание значений с помощью указателей

- При присваивании значения в область памяти, адресуемой указателем, его (указатель) можно использовать с левой стороны от оператора присваивания.
- Например, при выполнении следующей инструкции (если p — указатель на целочисленный тип)

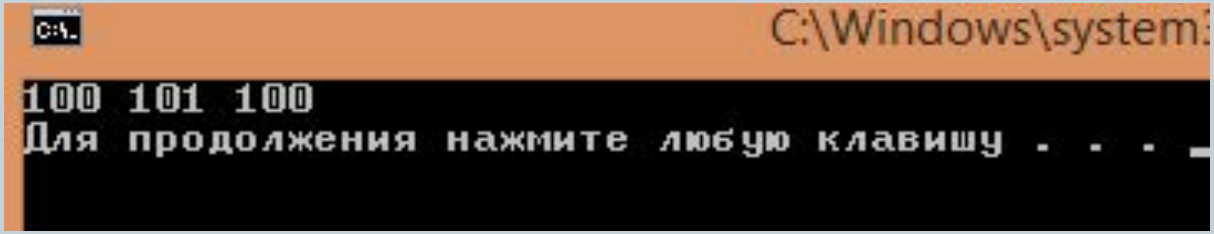
```
*p = 101;
```

Указатели


```
#include <iostream>
using namespace std;
int main()
{
    int *p, num=10;
    p = &num;
    *p = 100;
    cout << num << ' ';
    (*p)++;
    cout << num << ' ';
    (*p)--;
    cout << num << '\n';

    return 0;
}
```

Что будет выведено на экран?



Использование указателей в выражениях

- Указатели можно использовать в большинстве допустимых выражениях C++.
- С указателями можно использовать только четыре арифметических оператора:

  ++

  --

  +

  -

Использование указателей в выражениях

- Пусть p1 — указатель на int-переменную, которая располагается в памяти по адресу 2000. Над указателем выполняется следующая операция в 32-разрядной среде

```
p1++;
```

- Что будет в переменной-указателе p1?

Использование указателей в выражениях

- содержимое переменной-указателя p1 станет равным 2 004, а не 2 001!
- Дело в том, что при каждом инкрементировании указатель p1 будет указывать на следующее int-значение.
- Т.е. к адресу хранящемуся в указатели p1 добавиться число байт необходимое для хранения переменной типа int.

Использование указателей в выражениях

- Со значениями указателей можно выполнять операции сложения и вычитания, используя в качестве второго операнда целочисленные значения. Выражение

```
p1 = p1 + 9;
```

- заставляет p1 ссылаться на девятый элемент базового типа указателя p1 относительно элемента, на который p1 ссылался до выполнения этой инструкций.

Использование указателей в выражениях

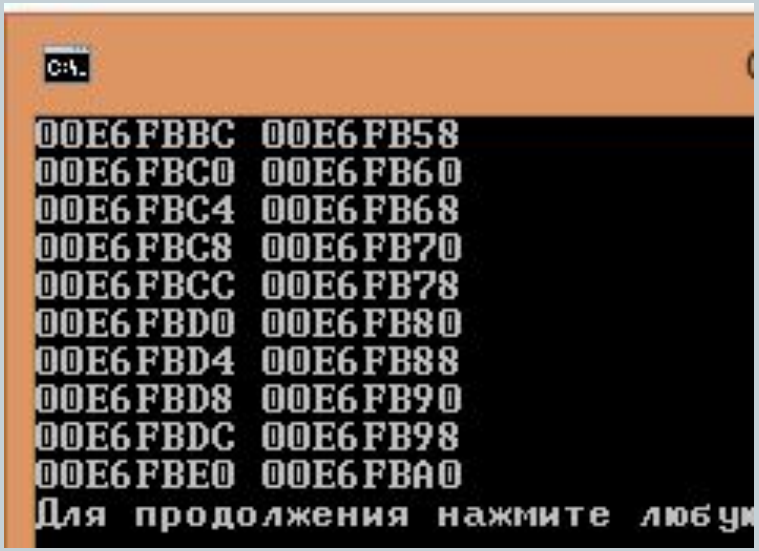
- Чтобы понять, как формируется результат выполнения арифметических операций над указателями, выполним следующую короткую программу. Она выводит реальные физические адреса, которые содержат указатель на int-значение (i) и указатель на float-значение (f).

Использование указателей в выражениях


```
// Демонстрация арифметических операций над указателями.
#include <iostream>
using namespace std;

int main()
{
   int *i, j[10];
   double *f, g[10];
   int x;
   i = j;
   f = g;
   for (x = 0; x<10; x++)
   cout << i + x << ' ' << f + x << '\n';

   return 0;
}
```



Сравнение указателей

- Указатели можно сравнивать, используя операторы отношения ==, < и >.
- Однако для того, чтобы результат сравнения указателей поддавался интерпретации, сравниваемые указатели должны быть каким-то образом связаны.

Указатели и массивы

- В C++ указатели и массивы тесно связаны между собой, причем настолько, что зачастую понятия "указатель" и "массив" взаимозаменяемы.
- Для начала рассмотрим следующий фрагмент программы.

```
char str[80];
char *p1;
p1 = str;
```

Указатели и массивы

- В C++ использование имени массива без индекса генерирует указатель на первый элемент этого массива.
- Таким образом, при выполнении присваивания p1 = str адрес str[0] присваивается указателю p1
- Указатель p1 можно использовать для доступа к элементам этого массива

Указатели и массивы

- Например, если нужно получить доступ к пятому элементу массива str, используйте одно из следующих выражений:

```
str[4]
```

```
*(p1 + 4)
```

Указатели и массивы

- Важно! Убедитесь лишний раз в правильности использования круглых скобок в выражении с указателями. В противном случае ошибку будет трудно отыскать, поскольку внешне программа может выглядеть вполне корректной. Если у вас есть сомнения в необходимости их использования, примите решение в их пользу — вреда от этого не будет.

Индексирование указателя


```
// Индексирование указателя подобно массиву.
#include <iostream>
#include <cctype>
using namespace std;

int main()
{
   char str[20] = "I love you";
   char *p;
   int i;
   p = str;
  // Индексируем указатель.
   for (i = 0; p[i]; i++) p[i] = toupper(p[i]);
   cout << p; // Отображаем строку.
   return 0;
}
```

Многоуровневая непрямая адресация

- Можно создать указатель, который будет ссылаться на другой указатель, а тот — на конечное значение.
- Эту ситуацию называют многоуровневой непрямой адресацией (multiple indirection) или использованием указателя на указатель.
- Например, следующее объявление сообщает компилятору о том, что balance — это указатель на указатель на значение типа int.

```
int **balance;
```

Многоуровневая непрямая адресация


```
#include <iostream>
using namespace std;
int main()
{
    int x, *p, **q;
    x = 10;
    p = &x;
    q = &p;
    cout << **q;
    return 0;
}
```

Динамическое выделение памяти

- Динамическое выделение памяти необходимо для эффективного использования памяти компьютера.
- В C++ операции new и delete предназначены для динамического распределения памяти компьютера. Операция new выделяет память из области свободной памяти, а операция delete высвобождает выделенную память.

Динамическое выделение памяти

- Выделяемая память, после её использования должна высвобождаться, поэтому операции new и delete используются парами.
- Пример

```
int *ptrvalue = new int;
delete ptrvalue;
```

Динамическое выделение памяти


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

int main()
{
    int *ptrvalue = new int;
    *ptrvalue = 9;
    cout << "ptrvalue = " << *ptrvalue << endl;
    delete ptrvalue;
    system("pause");
    return 0;
}
```

Создание динамических массивов

- Чаще всего операции new и delete применяются, для создания динамических массивов, а не для создания динамических переменных.
- Пример создания одномерного динамического массива.

```
float *ptrarray = new float[10];
delete[] ptrarray;
```

```
#include <iostream>
#include <ctime>
#include <iomanip>
using namespace std;

int main()
{
    srand(time(0));
    float *ptrarray = new float[10];
    for (int count = 0; count < 10; count++)
   ptrarray[count] = (rand() % 10 + 1) / float((rand() % 10 + 1));
    cout << "array = ";
    for (int count = 0; count < 10; count++)
   cout << setprecision(2) << ptrarray[count] << "    ";
    delete[] ptrarray; // высвобождение памяти
    cout << endl;
    system("pause");
    return 0;
}
```

Двумерный динамический массив

- Сначала объявляется указатель второго порядка float **ptrarray, который ссылается на массив указателей float* [2], где размер массива равен двум.

- После чего в цикле for каждой строке массива выделяется память под пять элементов. В результате получается двумерный динамический массив ptrarray[2][5].

```
float **ptrarray = new float*[2]; // две строки в массиве
for (int count = 0; count < 2; count++)
    ptrarray[count] = new float[5]; // и пять столбцов
```

Двумерный динамический массив

- Пример высвобождения памяти отводимой под двумерный динамический массив.

```
for (int count = 0; count < 2; count++)
    delete[] ptrarray[count];
```