top
КОМПЬЮТЕРНАЯ АКАДЕМИЯ

ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ C++

Урок № 13

Ссылки, операторы new и delete

Содержание

Материалы урока прикреплены к данному PDF-файлу. Для доступа к материалам, урок необходимо открыть в программе Adobe Acrobat Reader.

1. Константный указатель и указатель на константу

В прошлом уроке мы говорили с вами об указателях, арифметике указателей, связи указателей и массивов. Продолжим наше погружение в указатели и в этом уроке. Нам необходимо познакомиться с одной из самых любимых тем на всех собеседованиях по C++: константный указатель и указатель на константу. Начнем с константного указателя.

Константный указатель — указатель, чьё значение нельзя менять после инициализации. Это значит, что указатель в процессе своей жизни может указывать только на одну ячейку памяти и его нельзя перенаправить на другой адрес. Синтаксис объявления константного указателя:

```
тип_данных* const имя_указателя = адрес
```

Наличие const после * показывает, что мы объявляем константный указатель. При создании такого указателя его обязательно нужно проинициализировать адресом. Например:

```
int a = 12;
// константный указатель на целое
// указывает на a
int* const ptr = &a;

// 12 12
cout << *ptr << " " << a;
```

Попытка объявить константный указатель без инициализации приведет к ошибке на этапе компиляции.

```
// ошибка на этапе компиляции
// не указали адрес при создании указателя
int* const ptr;
```

Попытка перенаправить такой указатель на другой адрес также приведет к ошибке.

```
int a = 12;
int* const ptr = &a;
cout << *ptr << " " << a;
int b = 23;

/*
  Ошибка на этапе компиляции
  Нельзя перенаправлять константный указатель
*/
ptr = &b;
```

Можно ли через константный указатель менять значение по адресу, содержащемуся в нём? Конечно!

```
int a = 12;
int* const ptr = &a;

// 12 12
cout << *ptr << " " << a <<endl;

// изменяем значение в a
*ptr = 95;

// 95 95
cout << *ptr << " " << a;
```

В примере выше мы меняем значение переменной a через константный указатель ptr.

Теперь рассмотрим указатель на константу.

Указатель на константу не позволяет модифицировать значение по адресу, содержащемуся в нём. Синтаксис объявления:

```
const тип_данных* имя_указателя
```

Обратите внимание, что вам не нужно сразу инициализировать такой указатель адресом. Вы можете в процессе работы программы перенаправлять его на разные адреса. Главное правило, через него нельзя изменить значение, на которое он указывает. Рассмотрим на примере:

```
int a = 73;
/*
    Создали указатель на константу целого типа
*/

const int* ptr;

// Указатель указывает на a
ptr = &a;

// 73 73
cout << *ptr << " " << a<<endl;

int b = 88;

// Теперь указатель указывает на b
ptr = &b;

// 88 88
cout << *ptr << " " << b;
```

Обратите внимание, что ни a, ни b не являются константами в коде. Они будут рассматриваться, как неизменяемые значения, только, когда вы попробуете их изменить через указатель.

```
int a = 73;
/*
   Создали указатель на константу целого типа
*/
const int* ptr;

// Указатель указывает на a
ptr = &a;

/*
   Ошибка на этапе компиляции
   ptr - это указатель на константу
*/
*ptr = 12;
```

И самое удивительное, что вы можете создать константный указатель на константу. В этом случае через такой указатель вы сможете только читать и его необходимо инициализировать адресом при создании. Например:

```
int a = 7;
/*
   Константный указатель на константу
   ptr указывает на a
   Значение в ptr нельзя менять
   Через ptr нельзя изменять значения в a
*/
const int* const ptr = &a;
// 7 7
cout << *ptr << " " << a;
```

```
/*
   Вызовет ошибку на этапе компиляции, так как через ptr
   нельзя модифицировать a
   *ptr = 78;
*/

/*
   int b = 45;
   Вызовет ошибку на этапе компиляции, так как ptr
   нельзя перенаправлять ptr = &b;
*/
```

Обязательно проанализируйте приведенные примеры, чтобы до конца понять разницу между константным указателем и указателем на константу.

2. Общие сведения о ссылках

С этого урока мы начнем рассматривать другой механизм передачи параметров внутрь функций: с использованием ссылок. Мы уже знаем два способа передачи параметров внутрь функции: по значению, по указателю. В случае передачи по значению, передаётся копия переменной. В случае передачи по указателю, передаётся адрес переменной.

Использование указателей в качестве способа доступа к переменным таит в себе опасность — если был изменен адрес, хранящийся в указателе, то этот указатель больше не ссылается на нужное значение. Это означает, что мы не сможем изменить значение, так как адрес был изменен в указателе.

Язык C++ предлагает альтернативу для более удобного и безопасного доступа к переменным: использование ссылок. Объявив ссылочную переменную, можно создать объект, который, как указатель, ссылается на какое-то значение, но, в отличие от указателя, постоянно привязан к этому значению. И его нельзя перепривязать к другому значению. Таким образом, ссылка на значение всегда ссылается на это значение. Ссылку можно объявить следующим образом:

```
<имя типа>& <имя ссылки> = <выражение>;
```

или

```
<имя типа>& <имя ссылки>(<выражение>);
```

Ссылка является псевдонимом (вторым именем) для уже существующего объекта. Раз ссылка является другим именем уже существующего объекта, то в качестве инициализирующего объекта должно выступать имя некоторого объекта, уже расположенного в памяти. Обращение к ссылке приводят к обращению к тому объекту, на который ссылка ссылается. Если вы пытаетесь получить адрес ссылки, вам вернется адрес того объекта, на который ссылка ссылается. Проиллюстрируем это на конкретном примере:

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

int main()
{
   int ivar = 1234;  // Переменной присвоено значение.
   int *iptr = &ivar; // Указателю присвоен адрес ivar
   int &iref = ivar; // Ссылка ассоциирована с ivar
   int *p = &iref;   // Указателю присвоен адрес iref

   cout << "ivar = " << ivar << "\n";
   cout << "*iptr = " << *iptr << "\n";
   cout << "iref = " << iref << "\n";
   cout << "*p = " << *p << "\n";
   return 0;
}
```

Результат работы программы:

```
ivar = 1234
*iptr = 1234
iref = 1234
*p = 1234
```

Комментарии к программе. *Здесь объявляются четыре переменные. Переменная* ivar *инициализирована значением* 1234. *Указателю на целое* *iptr *присвоен адрес* ivar. *Переменная* iref *объявлена как ссылочная. Ссылка* iref *это псевдоним для переменной* ivar.

Оператор:

```
cout << "iref = " << iref << "\n";
```

выводит на экран значение переменной ivar. Это объясняется тем, что iref — ссылка на ivar.

Обращение к iref приводит к ivar. Последнее объявление int *p = &iref; создает еще один указатель, которому присваивается адрес, переменной на которую ссылается iref. В нашем случае это ivar. Строки:

```
int *iptr = &ivar;
```

и

```
int *p = &iref;
```

дают одинаковый результат. В них создаются указатели, указывающие на ivar. На рис. 1 проиллюстрирована взаимосвязь переменных из приведенной программы.

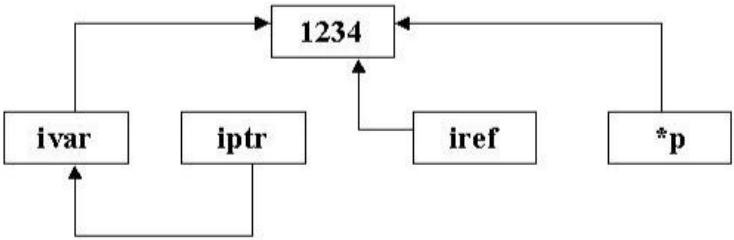

Рисунок 1

При использовании ссылок следует помнить одно правило: однажды инициализировав ссылку ей нельзя присвоить другое значение! Все эти конструкции:

```
a) int iv = 3;     b) iref++;     c) iref = 4321;
   iref = iv;
```

приведут к изменению переменной ivar!

Замечания:

1. В отличие от указателей, которые могут быть объявлены не инициализированными или установленными в нуль (0, NULL, nullptr), ссылки всегда ссылаются на объект. Для ссылок ОБЯЗАТЕЛЬНА инициализация при создании и не существует аналога нулевого указателя.
2. Не существует операторов, непосредственно производящих действия над ссылками! Это происходит, потому что ссылка — второе имя некоторого объекта. И все операции над ссылкой, это операции над этим объектом.
3. Стандарт C++ не определяет, как компилятор должен реализовать механизм ссылок. Некоторые заменяют обращение к ссылке обращением к реальной переменной. Некоторые заменяют ссылку на указатель, через который происходит обращение к переменной. Для того, чтобы не вступать в бессмысленные споры, мы будем считать, что ссылка — это псевдоним для переменной.

3. Ссылочные параметры. Передача аргументов по ссылке

Ссылочные переменные в чистом виде используются достаточно редко: значительно удобнее использовать саму переменную, чем ссылку на нее. В качестве параметров функции ссылки имеют более широкое применение. Ссылки особенно полезны в функциях, возвращающих несколько объектов (значений). Одно значение мы уже умеем возвращать с помощью return. А что же делать, если нам требуется изменить внутри функции два и более значений? Для решения этой задачи мы уже использовали механизм передачи переменной через указатель. Также эту задачу можно решить, используя механизм ссылок. Проиллюстрируем вышесказанное:

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

// Обмен с использованием указателей
void interchange_ptr(int* u, int* v) {
    int temp = *u;
    *u = *v;
    *v = temp;
}

// Обмен с использованием ссылок
void interchange_ref(int& u, int& v) {
    int temp = u;
```

```
    u = v;
    v = temp;
}

int main() {
    int x = 5, y = 10;
    cout << "Change with pointers:\n";
    cout << "x = " << x << "    y = " << y << "\n";

    // используем указатели
    interchange_ptr(&x, &y);
    cout << "x = " << x << "    y = " << y << "\n";
    cout << "-------------------------  " << "\n";
    cout << "Change with references:\n";
    cout << "x = " << x << "    y = " << y << "\n";

    // используем ссылки
    interchange_ref(x, y);
    cout << "x = " << x << " y = " << y << "\n";
    return 0;
}
```

В функции interchange_ptr() параметры описаны как указатели. Поэтому в теле этой функции выполняется их разыменование, а при обращении к этой функции в качестве фактических переменных используются адреса (&x, &y) тех переменных, значения которых нужно поменять местами. В функции interchange_ref() параметрами являются ссылки. Ссылки обеспечивают доступ из тела функции к фактическим параметрам, в качестве которых используются обычные переменные, определенные в программе. Обратите внимание насколько удобнее в функции использовать ссылки вместо указателей.

Рассмотрим ещё один пример передачи через указатель и ссылку. Ниже у нас небольшая функция:

```
void f(int *ip)
{
    *ip = 12;
}
```

Внутри этой функции осуществляется доступ к переданному аргументу, адрес которого хранится в указателе ip, с помощью следующего оператора:

```
f(&ivar); // Передача адреса ivar.
```

Внутри функции выражение *ip = 12; присваивает 12 переменной ivar, адрес которой был передан в функцию f(). Теперь рассмотрим аналогичную функцию, использующую ссылочные параметры:

```
void f(int &ir)
{
    ir = 12;
}
```

Указатель ip заменен ссылкой ir, которой присваивается значение 12. Выражение:

```
f(ivar); // Передача ivar по ссылке.
```

присваивает значение ссылочному объекту: передает ivar по ссылке функции f(). Поскольку ir ссылается на ivar, то ivar присваивается значение 12. Использование ссылочных параметров позволяет значительно повысить читабельность программы и уменьшить её сложность.

Теперь, когда мы познакомились с ссылками, перейдём к следующему разделу и рассмотрим одно из их предназначений.

4. Ссылки в качестве результатов функций

Здесь мы рассмотрим использование ссылок в качестве результатов функций.

Функции могут возвращать ссылки на объекты при условии, что эти объекты существуют, когда функция неактивна. Таким образом, функции не могут возвращать ссылки на свои локальные переменные. Например, для функции, объявленной как:

```
double &rf(int p);
```

необходим аргумент целого типа, и она возвращает ссылку на объект double, предположительно объявленный где-то в другом месте(например, на глобальном уровне).

Проиллюстрируем сказанное конкретными примерами.

Пример 1. Заполнение двумерного массива одинаковыми числами

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

int & rf(int index); // Прототип функции.
int a[10][2];
int main ()
{
    int b;
    cout << "Fill array.\n";
    for (int i=0;i<10;i++)
    {
```

```
        cout << i+1 << " element:";
        cin >> b;
        a[i][0] = b;
        rf(i) = b;
    }
    cout << "Show array.\n";
    cout << "1-st column 2-nd column" << "\n";
    for (int i=0; i<10; i++)
        cout << a[i][0] << "\t\t" << rf(i) << "\n";
    return 0;
}

int &rf(int index)
{
    return a[index][1]; // Возврат ссылки на элемент
                        // массива.
}
```

Здесь объявляется глобальный двумерный массив a, состоящий из целых чисел. В начале функции main() содержится прототип функции rf(), которая возвращает ссылку на целое значение второго столбца массива a. Оно однозначно идентифицируется параметром-индексом index. Так как функция rf() возвращает ссылку на целое значение, то имя функции может оказаться слева от оператора присваивания, что продемонстрировано в строке:

```
rf(i) = b;
```

Пример 2. Нахождение максимального элемента в массиве и замена его на нуль

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

```
// Функция определяет ссылку на элемент
// массива с максимальным значением.
int& rmax(int n, int d[])
{
    int im=0;
    for (int i=1; i<n; i++)
        im = d[im]>d[i]?im:i;
    return d[im];
}

int main ()
{
    int x[]={10, 20, 30, 14};
    int n=4;
    cout << "\nrmax(n,x) = " << rmax(n,x)
    cout << "\n";

    // заменяем максимум на ноль
    rmax(n,x) = 0;
    for (int i=0; i<n; i++)
        cout << "x[" << i << "]=" << x[i] << " ";
    cout << "\n";
    return 0;
}
```

Результаты работы программы:

```
rmax (n,x) = 30
x[0]=10    x[1]=20    x[2]=0    x[3]=14
```

При выполнении строки:

```
cout << "\nrmax(n,x) = " << rmax(n,x) << "\n";
```

происходит первое обращение к функции rmax(), первый аргумент которой — количество элементов в массиве,

а второй — сам массив. В результате возвращается ссылка на максимальный элемент массива, используя которую, это максимальное значение выводится на экран. При выполнении строки:

```
rmax(n,x) = 0;
```

снова осуществляется обращение к функции rmax(). Теперь уже по найденной ссылке максимальное значение меняется на 0.

Мы можем произвести замену значения, так как мы возвращаем ссылку на оригинальный элемент массива.

5. Операторы выделения памяти new и delete

Операция выделения памяти new

С помощью new мы можем себе позволить выделять память динамически — т.е. на этапе выполнения программы. Это значит, что у нас есть возможность спросить пользователя сколько ему нужно элементов в массиве и выделить память только под требуемое количество.

У new есть два формы использования. Первая — это динамическое выделение памяти под один объект некоторого типа. Синтаксис:

```
указатель_на_тип_= new имя_типа (инициализатор)
```

Инициализатор — это необязательное инициализирующее выражение, которое может использоваться для всех типов, кроме массивов.

При выполнении оператора

```
int *ip = new int;
```

создаются 2 объекта: динамический безымянный объект целого типа (фактически безымянная переменная целого типа) и указатель на него с именем ip, значением которого является адрес этого объекта. Можно создать и другой указатель на тот же динамический объект:

```
int *other=ip;
```

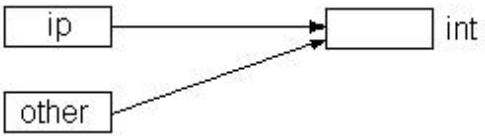

Рисунок 2

Через указатель ip можно модифицировать значение в выделенной ячейке.

```
*ip = 78;
cout <<*ip;
```

Если указателю ip присвоить другое значение, то можно потерять доступ к динамическому объекту:

```
int *ip=new int;
int i=0;
ip=&i;
```

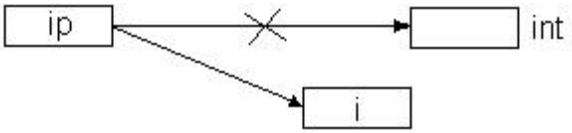

Рисунок 3

В результате динамический объект по-прежнему будет существовать, но обратиться к нему уже нельзя. Такие объекты называются мусором. Эта память будет занята до конца работы программы и это типичный пример утечки памяти.

При выделении памяти объект можно инициализировать:

```
int *ip = new int(3);
//3
cout<<*ip;
```

Вторая форма new используется для динамического выделения памяти под массив:

```
double *mas = new double [50];
```

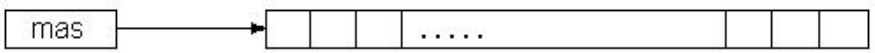

Рисунок 4

Далее с этой динамически выделенной памятью можно работать, как с обычным массивом:

```
#include<iostream>
#include<time.h>
using namespace std;

int main() {
    srand(time(NULL));
    int size;
    int* dar;
    // запрос размера массива с клавиатуры
    cout<<"Enter size:\n";
    cin >> size;
    /* выделение памяти под массив с количеством
       элементов size
    */
    dar = new int[size];
    // заполнение массива и показ на экран
    for (int i = 0; i < size; i++) {
        dar[i] = rand() % 100;
        cout << dar[i] << "\t";
```

```
    }
    cout << "\n\n";
    return 0;
}
```

В случае успешного завершения операция new возвращает адрес начала выделенного участка.

Операция освобождения памяти delete

Операция delete освобождает для дальнейшего использования в программе участок памяти, ранее выделенный операцией new:

```
// Удаляет динамический объект типа int
// если было ip = new int;
delete ip;
/* удаляет динамический массив длиной 50, если было
   double *mas = new double[50]; */
delete [] mas;
```

Совершенно безопасно применять операцию к нулевому указателю. Результат повторного применения операции delete к одному и тому же ненулевому указателю не определен. Поэтому, если вам указатель не нужен будет в дальнейшем лучше его обнулить, чтобы избежать неопределенного поведения.

Например так:

```
int *ip=new int[500];
...
if (ip){
    delete []ip;
```

```
    ip=nullptr;
}
else
{
    cout <<" delete was done before \n";
}
```

В наш, вышеописанный пример, мы теперь можем добавить освобождение памяти.

```
#include<iostream>
#include<time.h>
using namespace std;

int main() {
    srand(time(NULL));
    int size;
    int* dar;
    //    запрос размера массива с клавиатуры
    cout<<"Enter size:\n";
    cin >> size;
    /*
       выделение памяти под массив с количеством
       элементов size
    */
    dar = new int[size];
    // заполнение массива и показ на экран
    for (int i = 0; i < size; i++) {
        dar[i] = rand() % 100;
        cout << dar[i] << "\t";
    }
    cout << "\n\n";
    delete[] dar;
    return 0;
}
```

ПОДВЕДЕМ ИТОГ. *Если вы выполнили динамическое выделение памяти через new, не забудьте освободить память через delete.*

6. Домашнее задание

1. Даны два массива: A[M] и B[N] (M и N вводятся с клавиатуры). Необходимо создать третий массив минимально возможного размера, в котором нужно собрать элементы массива A, которые не включаются в массив B, без повторений.
2. Даны два массива: A[M] и B[N] (M и N вводятся с клавиатуры). Необходимо создать третий массив минимально возможного размера, в котором нужно собрать элементы массивов A и B, которые не являются общими для них, без повторений.
3. Даны два массива: A[M] и B[N] (M и N вводятся с клавиатуры). Необходимо создать третий массив минимально возможного размера, в котором нужно собрать элементы обоих массивов.
4. Дан массив: A[M] (M вводится с клавиатуры). Необходимо удалить из массива четные или нечетные значения. Пользователь вводит данные в массив, а также с помощью меню решает, что нужно удалить.