Лекция 3
Ссылки и указатели

Ссылки

В языке Си ссылок нет. Вместо ссылок в Си используют **указатели**.

В C++ ссылка (reference) представляет собой способ манипулировать каким-либо объектом. Фактически ссылка − это альтернативное имя для объекта. Для определения ссылки применяется знак амперсанда &:

```
int number {5};
int &refNumber {number};
```

В данном случае определена ссылка refNumber, которая ссылается на объект number. При этом в определении ссылки используется тот же тип, который представляет объект, на который ссылка ссылается, то есть в данном случае int.

Внимание! Просто определить ссылку нельзя:

```
int &refNumber;   //Ошибка
```

Она обязательно должна указывать на какой-нибудь объект.

Также нельзя присвоить ссылке литеральное значение, например, число:

```
int &refNumber = 10;   //Ошибка
```

После установления ссылки можно через нее манипулировать самим объектом, на который она ссылается:

```
#include <iostream>
int main()
{
   int number {5};
   int &refNumber {number};
   std::cout << refNumber << std::endl; // 5
   refNumber = 20;
   std::cout << number << std::endl;   // 20
}
```

Изменения по ссылке неизбежно скажутся и на том объекте, на который ссылается ссылка.

Можно определять не только ссылки на переменные, но и ссылки на константы. Но при этом ссылка сама должна быть константной:

```
const int number{5};
const int &refNumber{number};
std::cout << refNumber << std::endl; // 5
//refNumber = 20;      // изменять значение по ссылке нельзя
```

Инициализировать неконстантную ссылку константным объектом нельзя:

```
const int number {5};
int &refNumber {number};  // ошибка
```

Также константная ссылка может указывать и на обычную переменную, только значение по такой ссылке нельзя изменить:

```
int number {5};
const int &refNumber {number};
std::cout << refNumber << std::endl; // 5

//refNumber = 20;       //изменять значение по ссылке на константу нельзя

// но можно изменить саму переменную
number = 20;
std::cout << refNumber << std::endl;     //20
```

В данном случае несмотря на то, что нельзя напрямую изменить значение по константной ссылке, тем не менее можно изменить сам объект, что приведет естественно к изменению константной ссылки.

Ссылки в цикле for

В большинстве случае ссылки находят свое применение в функциях, когда надо передать значения по ссылке. Однако есть и другие сценарии использования ссылок. Например, в цикле for, который перебирает последовательность в стиле "for-each", нельзя изменить значения перебираемых элементов. Например:

```
#include <iostream>
 int main()
{
    int numbers[] {1, 2, 3, 4, 5};
    // меняем число на его квадрат
    for (auto n : numbers)
    {
        n = n * n;
    }
    // результат
    for (auto n : numbers)
    {
        std::cout << n << "\t";
    }
    std::cout << std::endl;
}
```

Здесь два цикла. В первом цикле при переборе массива помещаем каждый элемент массива в переменную *n* и изменяем ее значение на квадрат числа. Однако это приведет только к изменению этой переменной *n*, но никак не элементов перебираемого массива numbers. Элементы массива сохранят свои значения, что нам и покажет второй цикл, который выводит элементы на консоль:

```
1       2       3       4       5
```

Теперь используем ссылки:

```
#include <iostream>

int main()
{
    int numbers[] {1, 2, 3, 4, 5};
    // теперь n - ссылка на элемент массива
    for (auto& n : numbers)
    {
        n = n * n;
    }
    // результат
    for (auto n : numbers)
    {
        std::cout << n << "\t";
    }
    std::cout << std::endl;
}
```

Теперь в первом цикле переменная *n* представляет ссылку на элемент массива. Использование ссылки позволяет оптимизировать работу с циклом, поскольку теперь значение элемента массива не копируется в переменную *n*. И через ссылку можно изменить значение соответствующего элемента:

```
1       4       9       16      25
```

Иногда, наоборот, не нужно или даже нежелательно изменять элементы коллекции. В этом случае можно сделать ссылку константной:

```
#include <iostream>
 int main()
{
    int numbers[] {1, 2, 3, 4, 5};
    // n - константная ссылка
    for (const auto& n : numbers)
    {
```

```
        std::cout << n << "\t";
    }
    std::cout << std::endl;
}
```

Хотя здесь нельзя изменять значение элемента, но также с помощью ссылок оптимизируется перебор массива, так как элементы массива не копируются в переменную *n*.

Указатели

Указатели представляют собой объекты, значением которых служат адреса других объектов (переменных, констант, указателей) или функций. Как и ссылки, указатели применяются для косвенного доступа к объекту. Однако в отличие от ссылок указатели обладают большими возможностями.

Для определения указателя надо указать тип объекта, на который указывает указатель, и символ звездочки *:

тип_данных* название_указателя;

Сначала идет тип данных, на который указывает указатель, и символ звездочки *, затем имя указателя.

Например, указатель на объект типа int:

```
int* p;
```

Такой указатель может хранить только адрес переменной типа int, но пока данный указатель не ссылается ни на какой объект и хранит случайное значение. Его даже можно попробовать вывести на консоль:

```
#include <iostream>
int main()
{
    int* p;
    std::cout << p << std::endl;
}
```

В результате консоль выведет что-то типа "0x8" − некоторый адрес в шестнадцатеричном формате.

Также можно инициализировать указатель некоторым значением:

```
int* p{};
```

Поскольку конкурентное значение не указано, указатель в качестве значения получает число 0. Это значение представляет специальный адрес, который не указывает не на что.

Можно явным образом инициализировать нулем, например, используя специальную константу nullptr:

```
int* p{nullptr};
```

Никто не запрещает не инициализировать указатели, но в общем случае рекомендуется все-таки инициализировать, либо каким-то конкретным значением, либо нулем, как выше. Так, к примеру, нулевое значение в будущем позволит определить, что указатель не указывает ни на какой объект.

Положение звездочки не влияет на определение указателя: ее можно помещать ближе к типу данных, либо к имени переменной − оба определения будут равноценны:

```
int* p1{};
int *p2{};
```

Размер значения указателя (хранимый адрес) не зависит от типа указателя. Он зависит от конкретной платформы. На 32-разрядных платформах размер адресов равен 4 байтам, а на 64-разрядных − 8 байтам. Например:

```
#include <iostream>
int main()
{
   int *pint{};
   double *pdouble{};
   std::cout << "*pint size: " << sizeof(pint) << std::endl;
   std::cout << "*pdouble size: " << sizeof(pdouble) << std::endl;
}
```

Здесь определены два указателя на разные типы − int и double. Переменные этих типов имеют разные размеры − 4 и 8 байт соответственно. Но размеры значений указателей будут одинаковы. В случае 64-разрядной платформы размер обоих указателей равен 8 байтам.

Получение адреса и оператор &

С помощью операция & можно получить адрес некоторого объекта, например, адрес переменной. Затем этот адрес можно присвоить указателю:

```
int number {25};
int *pnumber {&number}; // указатель pnumber хранит адрес переменной number
```

Выражение &number возвращает адрес переменной number. Поэтому переменная pnumber будет хранить адрес переменной number. Что важно, переменная number имеет тип int, и указатель, который указывает на ее адрес, тоже имеет тип int. То есть должно быть соответствие по типу.

Можно использовать ключевое слово auto:

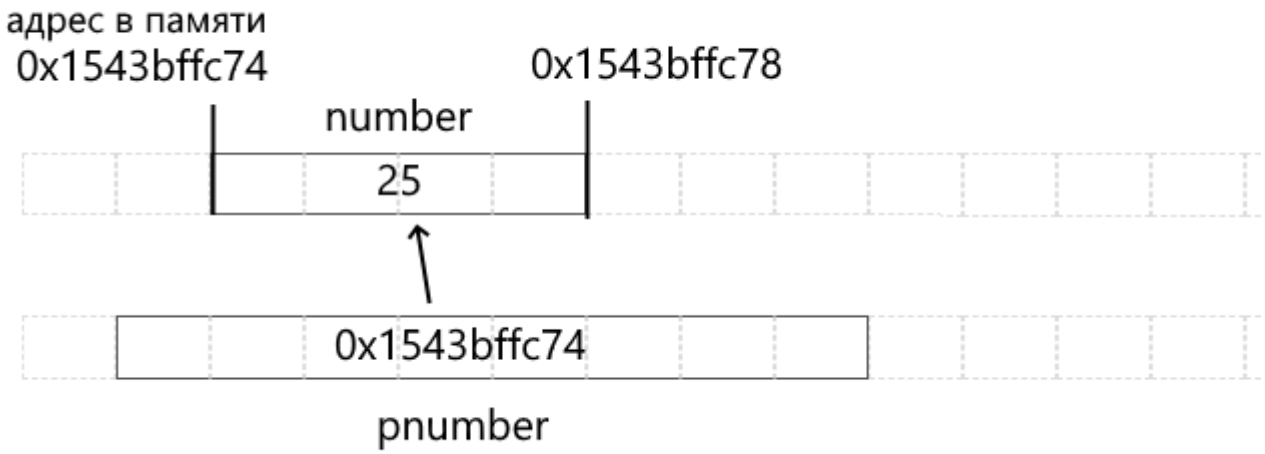
```
int number {25};
auto *pnumber {&number}; // указатель pnumber хранит адрес переменной number
```

Если попробовать вывести адрес переменной на консоль, то увидим, что он представляет шестнадцатеричное значение:

```
#include <iostream>
int main()
{
   int number {25};
   int *pnumber {&number}; // указатель pnumber хранит адрес переменной number
   std::cout << "number addr: " << pnumber << std::endl;
}
```

Консольный вывод программы:
number addr: 0x1543bffc74

В каждом отдельном случае адрес может отличаться и при разных запусках программы может меняться. Здесь машинный адрес переменной number – 0x1543bffc74. То есть в памяти компьютера есть адрес 0x1543bffc74, по которому располагается переменная number. Так как переменная представляет тип int, то на большинстве архитектур она будет занимать следующие 4 байта (на конкретных архитектурах размер памяти для типа int может отличаться). Таким образом, переменная типа int последовательно займет ячейки памяти с адресами 0x1543bffc74, 0x1543bffc75, 0x1543bffc76, 0x1543bffc77.



И указатель pnumber будет ссылаться на адрес, по которому располагается переменная number, то есть на адрес 0x1543bffc74.

Итак, указатель pnumber хранит адрес переменной number, а где хранится сам указатель pnumber? Чтобы узнать это, можно также применить к переменной pnumber операцию &:

```
#include <iostream>
int main()
{
   int number {25};
   int *pnumber {&number}; // указатель pnumber хранит адрес переменной number
   std::cout << "number addr: " << pnumber << std::endl;
   std::cout << "pnumber addr: " << &pnumber << std::endl;
}
```

Консольный вывод программы:
number addr: 0xe1f99ff7cc
pnumber addr: 0xe1f99ff7c0

Здесь переменная number располагается по адресу 0xe1f99ff7cc, а указатель, который хранит этот адрес − по адресу 0xe1f99ff7c0. Из вывода видно, что обе переменные хранятся совсем рядом в памяти

Получение значения по адресу

Так как указатель хранит адрес, то можно по этому адресу получить хранящееся там значение, то есть значение переменной number. Для этого применяется операция * или операция разыменования. Результатом этой операции всегда является объект, на который указывает указатель.

Применим данную операцию и получим значение переменной number:

```
#include <iostream>

int main()
{
   int number {25};
   int *pnumber {&number};
   std::cout << "Address = " << pnumber<< std::endl;
   std::cout << "Value = " << *pnumber << std::endl;
}
```

Вывод на консоль:
Address = 0x44305ffd4c
Value = 25

Значение, которое получено в результате операции разыменования, можно присвоить другой переменной:

```
int n1 {25};
int *pn1 {&n1}; // указатель pn1 хранит адрес переменной n1
int n2 { *pn1}; // n2 получает значение, которое хранится по адресу в pn1
std::cout << "n2 = " << n2 << std::endl;  // n2=25
```

И также используя указатель, можно менять значение по адресу, который хранится в указателе:

```
int x = 10;
int *px = &x;
*px = 45;
std::cout << "x = " << x << std::endl;     // 45
```

Так как по адресу, на который указывает указатель, располагается переменная *x*, то соответственно ее значение изменится.

Операции с указателями

Указатели поддерживают ряд операций: присваивание, получение адреса указателя, получение значения по указателю, некоторые арифметические операции и операции сравнения.

Присваивание адреса

Указателю можно присвоить адрес объекта того же типа, либо значение другого указателя. Для получения адреса объекта используется операция &:

```
int a {10};
int *pa {&a};   // указатель pa хранит адрес переменной a
```

При этом указатель и переменная должны иметь один и тот же тип, в данном случае это тип int.

Разыменование указателя

Операция разыменования указателя представляет выражение в виде *имя_указателя. Эта операция позволяет получить объект по адресу, который хранится в указателе.

```
#include <iostream>
int main()
{
    int a {10};
    int *pa {&a};   // хранит адрес переменной a

    std::cout << "*pa = " << *pa << std::endl;  // *pa = 10
    std::cout << "a = " << a << std::endl;      // a = 10

    *pa = 25;   // меняем значение по адресу в указателе

    std::cout << "*pa = " << *pa << std::endl;  // *pa = 25
    std::cout << "a = " << a << std::endl;      // a = 25
}
```

Через выражение **pa* можно получить значение по адресу, который хранится в указателе pa, а через выражение типа **pa = значение* вложить по этому адресу новое значение.

И так как в данном случае указатель *pa* указывает на переменную *a*, то при изменении значения по адресу, на который указывает указатель, также изменится и значение переменной *a*.

Присвоение указателю другого указателя

```
#include <iostream>
int main()
{
   int a {10};
   int b {2};

   int *pa {&a};   // указатель на переменную a
   int *pb {&b};   // указатель на переменную b

   std::cout << "pa: address=" << pa << "\t value=" << *pa << std::endl;
   std::cout << "pb: address=" << pb << "\t value=" << *pb << std::endl;

   pa = pb;    // теперь указатель pa хранит адрес переменной b
   std::cout << "pa: address=" << pa << "\t value=" << *pa << std::endl;
   *pa = 125;  // меняем значение по адресу в указателе pa
   std::cout << "b value=" << b << std::endl;
}
```

Когда указателю присваивается другой указатель, то фактически первый указатель начинает также указывать на тот же адрес, на который указывает второй указатель:

```
pa: address=0x56347ffc5c        value=10
pb: address=0x56347ffc58        value=2
pa: address=0x56347ffc58        value=2
b value=125
```

Нулевые указатели

Нулевой указатель (null pointer) − это указатель, который не указывает ни на какой объект. Если мы не хотим, чтобы указатель указывал на какой-то конкретный адрес, то можно присвоить ему условное нулевое значение. Для определения нулевого указателя можно инициализировать указатель нулем или константой nullptr:

```
int *p1{nullptr};
int *p2{};
```

Ссылки на указатели

Так как ссылка не является объектом, то нельзя определить указатель на ссылку, однако можно определить ссылку на указатель. Через подобную ссылку можно изменять значение, на которое указывает указатель или изменять адрес самого указателя:

```
#include <iostream>
int main()
{
   int a {10};
   int b {6};

   int *p{};          // указатель
   int *&pRef {p};     // ссылка на указатель
   pRef = &a;    // через ссылку указателю p присваивается адрес переменной a
   std::cout << "p value=" << *p << std::endl;   // 10
   *pRef = 70;   // изменяем значение по адресу, на который указывает указатель
   std::cout << "a value=" << a << std::endl;    // 70

   pRef = &b;          // изменяем адрес, на который указывает указатель
   std::cout << "p value=" << *p << std::endl;   // 6
}
```

Адрес указателя

Указатель хранит адрес переменной, и по этому адресу можно получить значение этой переменной. Но кроме того, указатель, как и любая переменная, сам имеет адрес, по которому он располагается в памяти. Этот адрес можно получить также через операцию &:

```
int a {10};
int *pa {&a};
std::cout << "address of pointer=" << &pa << std::endl;        // адрес указателя
std::cout << "address stored in pointer=" << pa << std::endl;  // адрес, который
                                                                хранится в указателе − адрес переменной a
std::cout << "value on pointer=" << *pa << std::endl;       // значение по адресу в
                                                                указателе − значение переменной a
```

Операции сравнения

К указателям могут применяться операции сравнения >, >=, <, <=,==, !=. Операции сравнения применяются только к указателям одного типа. Для сравнения используются номера адресов:

```
#include <iostream>
int main()
{
   int a {10};
```

```
    int b {20};
    int *pa {&a};
    int *pb {&b};
     if(pa > pb)
        std::cout << "pa (" << pa << ") is greater than pb ("<< pb << ")" << std::endl;
    else
        std::cout << "pa (" << pa << ") is less or equal pb ("<< pb << ")" << std::endl;
}
```

Консольный вывод:
pa (0xa9da5ffdac) is greater than pb (0xa9da5ffda8)

Приведение типов

Иногда требуется присвоить указателю одного типа значение указателя другого типа. В этом случае следует выполнить операцию приведения типов с помощью операции (тип_указателя *):

```
#include <iostream>

int main()
{
    char c {'N'};
    char *pc {&c};          // указатель на символ
    int *pd {(int *)pc};    // указатель на int
    void *pv {(void*)pc};   // указатель на void
    std::cout << "pv=" << pv << std::endl;
    std::cout << "pd=" << pd << std::endl;
}
```

Для преобразования указателя к другому типу в скобках перед указателем ставится тип, к которому надо преобразовать. Причем если мы не можем просто создать объект, например, переменную типа void, то для указателя это вполне будет работать. То есть можно создать указатель типа void.

Кроме того, указатель на тип char (char *pc {&c}) при выводе на консоль система будет интерпретировать как строку:

```
std::cout << "pc=" << pc << std::endl;
```

Поэтому если мы все-таки хотим вывести на консоль адрес, который хранится в указателе типа char, то этот указатель надо преобразовать к другому типу, например, к void* или к int*.

Арифметика указателей

Указатели могут участвовать в арифметических операциях (сложение, вычитание, инкремент, декремент). Однако сами операции производятся немного иначе, чем с числами. И многое здесь зависит от типа указателя.

К указателю можно прибавлять целое число, и также можно вычитать из указателя целое число. Кроме того, можно вычитать из одного указателя другой указатель.

Рассмотрим вначале операции инкремента и декремента и для этого возьмем указатель на объект типа int:

```
#include <iostream>
int main()
{
    int n{10};
    int *pn {&n};
    std::cout << "address=" << pn << "\tvalue=" << *pn << std::endl;

    pn++;
    std::cout << "address=" << pn << "\tvalue=" << *pn << std::endl;

    pn--;
    std::cout << "address=" << pn << "\tvalue=" << *pn << std::endl;
}
```

Операция инкремента ++ увеличивает значение на единицу. В случае с указателем увеличение на единицу будет означать увеличение адреса, который хранится в указателе, на размер типа указателя. То есть в данном случае указатель на тип int, а размер объектов int в большинстве архитектур равен 4 байтам. Поэтому увеличение указателя типа int на единицу означает увеличение адреса в указателе на 4.

Консольный вывод выглядит следующим образом:

```
address=0x81315ffd84    value=10
address=0x81315ffd88    value=828374408
address=0x81315ffd84    value=10
```

Здесь видно, что после инкремента значение указателя увеличилось на 4: с 0x81315ffd84 до 0x81315ffd88. А после декремента, то есть уменьшения на единицу, указатель получил предыдущий адрес в памяти. Фактически увеличение на единицу означает, что мы хотим перейти к следующему объекту в памяти, который находится за текущим и на который указывает указатель. А уменьшение на единицу означает переход назад к предыдущему объекту в памяти.

После изменения адреса можно получить значение, которое находится по новому адресу, однако это значение может быть неопределенным, как показано в примере выше.

В случае с указателем типа int увеличение/уменьшение на единицу означает изменение адреса на 4. Аналогично, для указателя типа short эти операции изменяли бы адрес на 2, а для указателя типа char на 1.

```
#include <iostream>
int main()
{
   double d {10.6};
   double *pd {&d};
   std::cout << "Pointer pd: address:" << pd << std::endl;
   pd++;   // увеличение адреса на 8 байт − размер double
   std::cout << "Pointer pd: address:" << pd << std::endl;

   short n {5};
   short *pn {&n};
   std::cout << "Pointer pn: address:" << pn << std::endl;
   pn++;   // увеличение адреса на 2 байта − размер short
   std::cout << "Pointer pn: address:" << pn << std::endl;
}
```

Консольный вывод будет выглядеть следующим образом:

```
Pointer pd: address:0x2731bffd58
Pointer pd: address:0x2731bffd60

Pointer pn: address:0x2731bffd56
Pointer pn: address:0x2731bffd58
```

Как видно из консольного вывода, увеличение на единицу указателя типа double привело к увеличению хранимого в нем адреса на 8 единиц (размер объекта double − 8 байт), а увеличение на единицу указателя типа short дало увеличение хранимого в нем адреса на 2 (размер типа short − 2 байта).

Аналогично указатель будет изменяться при прибавлении/вычитании не единицы, а какого-то другого числа.

```
#include <iostream>

int main()
{
   double d {10.6};
   double *pd {&d};
   std::cout << "Pointer pd: address:" << pd << std::endl;
   pd = pd + 2;   // увеличение адреса на 16 байт − 2 объекта double
   std::cout << "Pointer pd: address:" << pd << std::endl;
```

```
    short n {5};
    short *pn {&n};
    std::cout << "Pointer pn: address:" << pn << std::endl;
    pn = pn - 3;   // уменьшение адреса на 6 байт − размер 3 объектов short
    std::cout << "Pointer pn: address:" << pn << std::endl;
}
```

Добавление к указателю типа double числа 2

pd = pd + 2;

означает, что мы хотим перейти на два объекта double вперед, что подразумевает изменение адреса на 2 * 8 = 16 байт.

Вычитание из указателя типа short числа 3

pn = pn - 3;

означает, что мы хотим перейти на три объекта short назад, что подразумевает изменение адреса на 3 * 2 = 6 байт.

Консольный вывод:
Pointer pd: address:0xb88d5ffbe8
Pointer pd: address:0xb88d5ffbf8

Pointer pn: address:0xb88d5ffbe6
Pointer pn: address:0xb88d5ffbe0

В отличие от сложения операция вычитания может применяться не только к указателю и целому числу, но и к двум указателям одного типа:

```
#include <iostream>

int main()
{
    int a{10};
    int b{23};
    int *pa {&a};
    int *pb {&b};
    auto ab {pa - pb};

    std::cout << "pa: " << pa << std::endl;
    std::cout << "pb: " << pb << std::endl;
    std::cout << "ab: " << ab << std::endl;
}
```

Согласно стандарту разность указателей представляет тип std::ptrdiff_t, который в реальности является псевдонимом для типов int, long и long long. Какой

конкретно из этих типов применяется для хранения разности, зависит от конкретной платформы. Например, на Windows 64x это тип long long. Поэтому переменная ab, которая хранит разность адресов, определена с помощью оператора auto.

Консольный вывод:
pa: 0x6258fffab4
pb: 0x6258fffab0
ab: 1

Результатом разности двух указателей является "расстояние" между ними. Например, в случае выше адрес из первого указателя на 4 больше, чем адрес из второго указателя (0x6258fffab0 + 4 = 0x6258fffab4). Так как размер одного объекта int равен 4 байтам, то расстояние между указателями будет равно (0x6258fffab4 - 0x6258fffab0)/4 = 1.

Некоторые особенности операций

При работе с указателями надо отличать операции с самим указателем и операции со значением по адресу, на который указывает указатель.

```
int a {10};
int *pa {&a};
int b {*pa + 20};   // операция со значением, на который указывает указатель
pa++;               // операция с самим указателем
std::cout << "b: " << b << std::endl;  ;  // 30
```

То есть в данном случае через операцию разыменования *pa получаем значение, на которое указывает указатель pa, то есть число 10, и выполняем операцию сложения. Таким образом, это обычная операция сложения между двумя числами, так как выражение *pa представляет число.

Но в то же время есть особенности, в частности, с операциями инкремента и декремента. Дело в том, что операции *, и префиксные операции ++ и -- имеют одинаковый приоритет и при размещении рядом выполняются справа налево.

Например, выполним префиксный инкремент:

```
int a {10};
int *pa {&a};
std::cout << "pa: address=" << pa << "\tvalue=" << *pa << std::endl;
int b {++*pa};      // инкремент значения по адресу указателя

std::cout << "b: value=" << b << std::endl;
std::cout << "pa: address=" << pa << "\tvalue=" << *pa << std::endl;
```

В выражении b {++*pa}; сначала происходит разыменовывание указателя, и мы получаем значение по его адресу, то есть число 10. Затем к этому числу прибавляется единица.

Результат работы может быть следующий:

```
pa: address=0x7ff7b31bd8b8    value=10
b: value=11
pa: address=0x7ff7b31bd8b8    value=11
```

Изменим выражение:

```
int b{*++pa};      // инкремент адреса указателя с последующим разыменовыванием
```

Теперь сначала к указателю прибавляется единица (то есть к адресу добавляется 4, так как указатель типа int), затем мы получаем по этому адресу значение и присваиваем его переменной b. Полученное значение в этом случае может быть неопределенным:

```
pa: address=0x7ff7b13d78b8    value=10
b: value=0
pa: address=0x7ff7b13d78bc    value=0
```

В отличие от префиксных инкремента и декремента постфиксные версии операций имеют больший приоритет, нежели операция разыменования *. Например, возьмем следующую программу:

```
int a {10};
int *pa {&a};
std::cout << "pa: address=" << pa << "\tvalue=" << *pa << std::endl;
int b{*pa++};      // инкремент адреса указателя с последующим разыменовыванием

std::cout << "b: value=" << b << std::endl;
std::cout << "pa: address=" << pa << "\tvalue=" << *pa << std::endl;
```

Поскольку постфиксный инкремент имеет больший приоритет, то в выражении *pa++ сначала увеличиваем адрес указателя pa на единицу (опять фактически на 4, так как указатель типа int) и затем получаем значение по адресу. Однако поскольку постфиксный инкремент возвращает значение до увеличения, то в переменную b мы получим значение, которое было по адресу до инкремента. Например, консольный вывод:

```
pa: address=0x7ff7b55288b8    value=10
b: value=10
pa: address=0x7ff7b55288bc    value=0
```

Изменим выражение:

```
b {(*pa)++};
```

Скобки изменяют порядок операций. Здесь сначала выполняется операция разыменования и получение значения, затем это значение увеличивается на 1. Теперь по адресу в указателе находится число 11. И затем, так как инкремент

постфиксный, переменная b получает значение, которое было до инкремента, то есть опять число 10. Таким образом, в отличие от предыдущего случая все операции производятся над значением по адресу, который хранит указатель, но не над самим указателем. И, следовательно, изменится результат работы:

```
pa: address=0x7ff7b7b268b8    value=10
b: value=10
pa: address=0x7ff7b7b268b8    value=11
```

Константы и указатели

Указатели на константы

Указатели могут указывать как на переменные, так и на константы. Чтобы определить указатель на константу, он тоже должен объявляться с ключевым словом const:

```
#include <iostream>

int main()
{
    const int a {10};
    const int *pa {&a};
    std::cout << "address=" << pa << "\tvalue=" << *pa << std::endl;
}
```

Здесь указатель pa указывает на константу a. Поэтому даже если мы захотим изменить значение по адресу, который хранится в указателе, мы не сможем это сделать, например так:

```
*pa = 34;
```

В этом случае мы просто получим ошибку во время компиляции.

Возможна также ситуация, когда указатель на константу на самом деле указывает на переменную:

```
#include <iostream>
int main()
{
    int a {10};
    const int *pa {&a};
    std::cout <<"value=" << *pa << std::endl;     // value=10
    a = 22;
    std::cout <<"value=" << *pa << std::endl;     // value=22
    //*pa = 34;     // так делать нельзя
}
```

В этом случае переменную отдельно можно изменить, однако по-прежнему изменить ее значение через указатель не получится.

Через указатель на константу нельзя изменять значение переменной/константы. Но можно присвоить указателю адрес любой другой переменной или константы:

```
const int a {10};
const int *pa {&a};     // указатель указывает на константу a
const int b {45};
pa = &b;                // указатель указывает на константу b
std::cout <<"*pa = " << *pa << std::endl;     // *pa = 45
std::cout <<"a = " << a << std::endl;     // a = 10 − константа a не изменяется
```

Константный указатель

От указателей на константы надо отличать константные указатели. Они не могут изменять адрес, который в них хранится, но могут изменять значение по этому адресу.

```
#include <iostream>

int main()
{
   int a {10};
   int *const pa {&a};
   std::cout << "value=" << *pa << std::endl;      // value = 10
   *pa = 22;                                  // меняем значение
   std::cout << "value=" << *pa << std::endl;      // value = 22

   int b {45};
   // pa = &b;      //   так нельзя сделать
}
```

Константный указатель на константу

И объединение обоих предыдущих случаев − константный указатель на константу, который не позволяет менять ни хранимый в нем адрес, ни значение по этому адресу:

```
int main()
{
   int a {10};
   const int *const pa {&a};

   //*pa = 22;  //так сделать нельзя

   int b {45};
```

```
    // pa = &b;  //так сделать нельзя
}
```