**Применение ключевого слова unsafe**

Любой код, в котором используются указатели, ***должен*** быть помечен как небезопасный

с помощью специального ключевого слова ***unsafe***. Подобным образом можно

пометить конкретные типы данных (например, классы и структуры), члены класса

(в том числе методы и операторы) или отдельные кодовые блоки как небезопасные.

В качестве примера ниже приведена программа, где указатели используются в методе

Main(), помеченном как небезопасный.

Сначала в настройках проекта, надо в билде поставить галочку для разрешения небезопасного мода.

(***glava20\_1***)

class UnsafeCode

{

//mark Main as unsafe

unsafe static void Main()

{

int count = 99;

int\* p; //pointer

p = &count; //address of count in p

Console.WriteLine("Value of count: " + \*p);

\*p = 10; //define count with 10

Console.WriteLine("New value of count: " + \*p);

}

}

**Применение модификатора fixed**

В работе с указателями нередко используется модификатор fixed, который препятствует

удалению управляемой переменной средствами "сборки мусора". Потребность

в этом возникает, например, в том случае, если указатель обращается к полю в объекте

определенного класса. А поскольку указателю ничего не известно о действиях системы

"сборки мусора", то он будет указывать не на тот объект, если удалить нужный объект.

Ниже приведена общая форма модификатора fixed:

fixed (тип\* р = &фиксированный\_объект) (

// использовать фиксированный объект

}

где *р* обозначает указатель, которому присваивается адрес объекта. Этот объект будет

оставаться на своем текущем месте в памяти до конца выполнения кодового блока.

В качестве адресата оператора fixed может быть также указано единственное выражение,

а не целый кодовый блок. Модификатор fixed допускается использовать только

в коде, помеченном как небезопасный. Кроме того, несколько указателей с модификатором

fixed могут быть объявлены списком через запятую.

Ниже приведен пример применения модификатора fixed.

(***glava20\_2***)

class Test

{

public int num;

public Test(int i) { num = i; }

}

class FixedCode

{

//mark main as unsafe

unsafe static void Main()

{

Test o = new Test(19);

fixed(int \* p = &o.num)//use fixed modificator to place

{ //address of o.num in p

Console.WriteLine("Value of o.num: " + \*p);

\*p = 10;

Console.WriteLine("New value of o.num: " + \*p);

}

}

}

В данном примере модификатор fixed препятствует удалению объекта о. А поскольку

переменная р указывает на переменную экземпляра о.num, то она будет указывать

на недостоверную область памяти, если удалить объект о.

**Доступ к членам структуры с помощью указателя**

Указатель может указывать на объект типа структуры при условии, что структура

не содержит ссылочные типы данных. Для доступа к члену структуры с помощью указателя

следует использовать оператор-стрелку (->), а не оператор-точку (.).

struct MyStruct

{

public int a;

public int b;

public int Sum() { return a + b; }

}

MyStruct о = new MyStruct();

MyStruct\* p; // объявить указатель

p = &o;

p->a = 10; // использовать оператор ->

p->b = 20; // использовать оператор ->

Console.WriteLine("Сумма равна " + p->Sum());

**Арифметические операции над указателями**

Над указателями можно выполнять только четыре арифметические операции: ++,

--, + и -. Для того чтобы стало понятнее, что именно происходит в арифметических

операциях над указателями, рассмотрим сначала простой пример. Допустим, что переменная

p1 является указателем с текущим значением 2000, т.е. она содержит адрес

2000. После выполнения выражения

***p1++;***

переменная p1 будет содержать значение 2004, а не 2001! Дело в том, что после каждого

инкрементирования переменная p1 указывает на следующее значение типа int. А поскольку

тип int представлен в C# 4 байтами, то в результате инкрементирования значение

переменной p1 увеличивается на 4. Справедливо и обратное: при каждом декрементировании

переменной p1 ее значение уменьшается на 4.

Все сказанное выше можно обобщить: после каждого инкрементирования указатель

будет указывать на область памяти, где хранится следующий элемент его соотносимого

типа, а после каждого декрементирования указатель будет указывать на область

памяти, где хранится предыдущий элемент его соотносимого типа.

Арифметические операции над указателями не ограничиваются только инкрементированием

и декрементированием. К указателям можно добавлять и вычитать из них

целые значения. Так, после вычисления следующего выражения:

***p1 = p1 + 9;***

переменная p1 будет указывать на девятый элемент ее соотносимого типа по отношению

к элементу, на который она указывает в настоящий момент.

Если складывать указатели нельзя, то разрешается вычитать один указатель из другого,

при условии, что оба указателя имеют один и тот же соотносимый тип. Результатом

такой операции окажется количество элементов соотносимого типа, которые

разделяют оба указателя.

Кроме сложения и вычитания целого числа из указателя, а также вычитания двух

указателей, другие арифметические операции над указателями не разрешаются. В частности,

к указателям нельзя добавлять или вычитать из них значения типа float или

double. Не допускаются также арифметические операции над указателями типа void\*.

Bыполните приведенную ниже короткую программу, где выводятся физические

адреса, на которые указывает целочисленный указатель (ip) и указатель с плавающей

точкой одинарной точности (fp). Понаблюдайте за изменениями каждого из

этих указателей по отношению к их соотносимым типам на каждом шаге цикла.

(***glava20\_3***)

class PtrArithDemo

{

unsafe static void Main()

{

int x;

int i;

double d;

int\* ip = &i;

double\* fp = &d;

Console.WriteLine("int double\n");

for (x = 0; x < 10; x++)

{

Console.WriteLine((uint)(ip) + " " + (uint)(fp));

ip++;

fp++;

}

}

}

Как следует из приведенного выше результата, арифметические операции выполняются

над указателями относительно их соотносимого типа. Так, значения типа int

занимают в памяти 4 байта, а значения типа double — 8 байтов, и поэтому их адреса

изменяются с приращением именно на эти величины.

**Сравнение указателей**

Указатели можно сравнивать с помощью таких операторов отношения, как ==, <

и >. Но для того чтобы результат сравнения указателей оказался содержательным, оба

указателя должны быть каким-то образом связаны друг с другом. Так, если переменные

p1 и р2 являются указателями на две разные и не связанные вместе переменные,

то любое их сравнение, как правило, не имеет никакого смысла. Но если переменные

p1 и р2 указывают на связанные вместе переменные, например на элементы одного

массива, то их сравнение может иметь определенный смысл.

**Указатели и массивы**

В C# указатели и массивы связаны друг с другом. Например, при указании имени

массива без индекса в операторе с модификатором fixed формируется указатель на

начало массива. В качестве примера рассмотрим следующую программу.

(***glava20\_4***)

class PrtArray

{

unsafe static void Main()

{

int[] nums = new int[10];

fixed(int \* p = & nums[0], p2 = nums)

{

if (p == p2)

Console.WriteLine("p and p2 has the same address.");

}

}

}

**Индексирование указателей**

Когда указатель обращается к массиву, его можно индексировать как сам массив.

Такой синтаксис служит более удобной в некоторых случаях альтернативой арифметическим

операциям над указателями. Рассмотрим следующий пример программы.

(***glava20\_5***)

class PtrIndexArray

{

unsafe static void Main()

{

int[] nums = new int[10];

fixed (int\* p = nums)

{

for (int i = 0; i < 10; i++)

p[i] = i;

for (int i = 0; i < 10; i++)

Console.WriteLine("p[{0}]: {1} ", i, p[i]);

}

Console.WriteLine("Using arithmetic operations with pointers.");

fixed (int\* p = nums)

{

for (int i = 0; i < 10; i++)

\*(p + i) = i;

for (int i = 0; i < 10; i++)

Console.WriteLine("\*(p + {0}): {1} ", i, \*(p + i));

}

}

}

Что касается индексирования указателей, то необходимо иметь в виду следующее.

Во-первых, при таком индексировании контроль границ массива не осуществляется.

Поэтому указатель может обращаться к элементу вне границ массива. И во-вторых,

для указателя не предусмотрено свойство Length, определяющее длину массива. Поэтому,

если используется указатель, длина массива заранее неизвестна.

**Указатели и строки**

Символьные строки реализованы в C# в виде объектов. Тем не менее отдельные символы

в строке могут быть доступны по указателю. Для этого указателю типа char\*

присваивается адрес начала символьной строки в следующем операторе с модификатором

fixed.

fixed (char\* р = str) { //

После выполнения оператора с модификатором fixed переменная *р* будет указывать

на начало массива символов, составляющих строку. Этот массив оканчивается

символом конца строки, т.е. нулевым символом. Поэтому данное обстоятельство можно

использовать для проверки конца массива. В С/С++ строки реализуются в виде массивов,

оканчивающихся символом конца строки, а следовательно, получив указатель

типа char\* на строку, ею можно манипулировать таким же образом, как и в C/C++.

Ниже приведена программа, демонстрирующая доступ к символьной строке по

указателю типа char\*.

(***glava20\_6***)

class FixedString

{

unsafe static void Main()

{

string str = "This is a test";

fixed (char\* p = str)

{

//output containing using pointer

for (int i = 0; p[i] != 0; i++)

Console.Write(p[i]);

}

Console.WriteLine();

}

}

**Многоуровневая непрямая адресация**

Один указатель может указывать на другой, а тот, свою очередь, — на целевое значение.

Это так называемая *многоуровневая непрямая адресация,* или применение *указателей*

*на указатели.* Такое применение указателей может показаться, на первый взгляд,

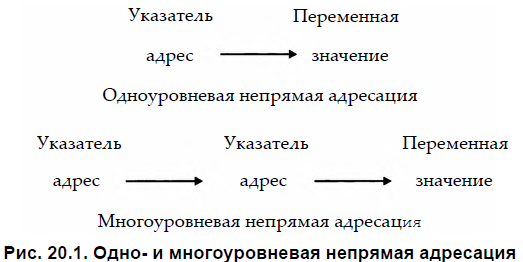
запутанным. Для прояснения принципа многоуровневой непрямой адресации обратимся

за помощью к рис. 20.1. Как видите, значением обычного указателя является

адрес переменной, содержащей требуемое значение. Если же применяется указатель

на указатель, то первый из них содержит адрес второго, указывающего на переменную,

содержащую требуемое значение.



Многоуровневая непрямая адресация может быть продолжена до любого предела,

но потребность более чем в двух уровнях адресации по указателям возникает крайне

редко. На самом деле чрезмерная непрямая адресация очень трудно прослеживается

и чревата ошибками.

Переменная, являющаяся указателем на указатель, должна быть объявлена как таковая.

Для этого достаточно указать дополнительный знак \* после имени типа переменной.

Например, в следующем объявлении компилятор уведомляется о том, что

переменная q является указателем на указатель и относится к типу int.

int\*\* q;

Для доступа к целевому значению, косвенно адресуемому по указателю на указатель,

следует дважды применить оператор \*, как в приведенном ниже примере.

(***glava20\_7***)

class MultipleIndirect

{

unsafe static void Main()

{

int x;

int\* p;

int\*\* q;

x = 10;

p = &x;

q = &p;

Console.WriteLine(\*\*q);

}

}

Результатом выполнения этой программы будет выведенное на экран значение 10

переменной х. В данной программе переменная р объявляется как указатель на значение

типа int, а переменная q — как указатель на указатель типа int.

И последнее замечание: не путайте многоуровневую непрямую адресацию со

структурами данных высокого уровня, в том числе связными списками, так как это совершенно

разные понятия.

**Массивы указателей**

Указатели могут быть организованы в массивы, как и любой другой тип данных.

Ниже приведен пример объявления массива указателей типа int длиной в три элемента.

int\*[] ptrs = new int\*[3];

Для того чтобы присвоить адрес переменной var типа int третьему элементу массива

указателей, достаточно написать следующую строку кода.

ptrs[2] = Svar;

А для того чтобы обнаружить значение переменной var, достаточно написать приведенную

ниже строку кода.

\* ptrs[2]

**Оператор sizeof**

Во время работы с небезопасным кодом иногда полезно знать размер в байтах

одного из встроенных в C# типов значений. Для получения этой информации служит

оператор sizeof. Ниже приведена его общая форма:

sizeof(тип)

где *тип* обозначает тот тип, размер которого требуется получить. Вообще говоря, оператор

sizeof предназначен главным образом для особых случаев и, в частности, для

работы со смешанным кодом: управляемым и неуправляемым.

**Оператор stackalloc**

Для распределения памяти, выделенной под стек, служит оператор stackalloc.

Им можно пользоваться лишь при инициализации локальных переменных. Ниже

приведена общая форма этого оператора:

тип\* р = stackalloc тип[размер]

где *р* обозначает указатель, получающий адрес области памяти, достаточной для хранения

объектов, имеющих указанный *тип,* в количестве, которое обозначает *размер.*

Если же в стеке недостаточно места для распределения памяти, то генерируется исключение

System.StackOverflowException. И наконец, оператор stackalloc можно

использовать только в небезопасном коде.

Как правило, память для объектов выделяется из кучи — динамически распределяемой

свободной области памяти. А выделение памяти из стека является исключением.

Ведь переменные, располагаемые в стеке, не удаляются средствами "сборки мусора",

а существуют только в течение времени выполнения метода, в котором они объявляются.

После возврата из метода выделенная память освобождается. Преимущество применения

оператора stackalloc заключается, в частности, в том, что в этом случае не

нужно беспокоиться об очистке памяти средствами "сборки мусора".

Ниже приведен пример применения оператора stackalloc.

(***glava20\_8***)

class UseStackAlloc

{

unsafe static void Main()

{

int\* ptrs = stackalloc int[3];

ptrs[0] = 1;

ptrs[1] = 2;

ptrs[2] = 3;

for (int i = 0; i < 3; i++)

Console.WriteLine(ptrs[i]);

}

}