Долгожданная адресная арифметика

Практический курс по C/C++: https://stepik.org/course/193691

Смотреть материал на YouTube | RuTube

Здравствуйте, дорогие друзья! Я Сергей Балакирев и мы продолжаем тему указателей. На этом занятии поговорим о том, чем пугают студентов информационных специальностей, из-за чего они не спят ночами или просыпаются в холодном поту. Да, это моя любимая и долгожданная адресная арифметика! Суть которой можно выразить одной фразой:

> ***Значение указателя меняется на размер типа данных, для которого он определен.***

Осталось только конкретизировать эту фразу и привести примеры.

Пусть в нашей программе объявлена целочисленная переменная g с начальным значением 4 и указатель ptr, который инициализирован на переменную g:

```
#include <stdio.h>

int main(void)
{
        int g = 4;
        int *ptr = &g;

        printf("ptr = %u\n", ptr);

        return 0;
}
```

С помощью функции printf() выполняется вывод адреса указателя ptr в виде беззнакового целого десятичного числа. После запуска программы увидим значение:

```
ptr = 6487832
```

Это номер ячейки, начиная с которой располагается целочисленная переменная g. А теперь давайте посмотрим, что будет, если значение адреса указателя ptr увеличить на единицу:

```
ptr++;
printf("ptr = %u\n", ptr);
```

Запустим программу, увидим значения:

```
ptr = 6487832
ptr = 6487836
```

И вот здесь некоторые начинающие программисты испытывают то, что называют мудреной фразой «когнитивный диссонанс». Почему операция инкремента увеличивает значение адреса сразу на четыре, а не на один, как это, возможно, ожидалось? Ответ очень прост. Когда мы работаем с указателями, а не с обычными переменными, то целочисленные арифметические операции выполняются в соответствии с правилами адресной арифметики. В частности, увеличение на единицу означает, что нам нужно перейти к следующей порции данных в памяти компьютера, а не к следующей ячейке. Именно поэтому адрес указателя увеличивается на размер типа данных, для которого он объявлен. В нашем примере – это тип int, который занимает 4 байта. Поэтому увеличивая на единицу значение адреса указателя ptr, мы получаем прибавку на эти четыре байта. Отсюда и получается такой результат.

Подобный результат будет, если выполнить операцию декремента:

```
ptr--;
printf("ptr = %u\n", ptr);
```

тогда значение адреса, наоборот, уменьшится на четыре:

```
ptr = 6487832
ptr = 6487828
```

Вот это и есть магия адресной арифметики. Вообще, мы можем с указателями выполнять следующие целочисленные арифметические операции:

```
#include <stdio.h>

int main(void)
{
        int g = 4;
        int *ptr = &g;

        printf("%p\n", ptr);

        ptr += 3;
        ptr -= 4;
        ptr = ptr + 10;
        ptr = ptr - 9;
        --ptr;
        ptr++;

        printf("%p\n", ptr);

        return 0;
}
```

Обратите внимание, операции должны быть именно целочисленными. В них должны фигурировать или целочисленные литералы или целочисленные переменные. Другие типы здесь использовать недопустимо. Также нельзя использовать операции умножения и деления. Исключение составляет только одна операция вычисления разности между двумя указателями, когда в операндах не используются целочисленные типы. Причем эта операция имеет смысл исключительно с элементами одного массива. Давайте посмотрим, как она работает на следующем примере:

```
#include <stdio.h>

int main(void)
{
    short ar[10];
    short *ptr = ar;
    short *p = &ar[3];

    printf("ptr = %u, p = %u\n", ptr, p);

    int res = p - ptr;

    printf("res = %d\n", res);

    return 0;
}
```

Смотрите, здесь объявлен массив ar и два указателя с инициализацией на адрес первого элемента и третьего элемента массива ar. После этого вычисляется разность между указателями p и ptr и результат заносится в обычную целочисленную переменную res.

После запуска программы увидим результат:

```
ptr = 6487808, p = 6487814
res = 3
```

Почему видим значение 3 в переменной res? Конечно, здесь выполняется адресная арифметика, которая возвращает расстояние в памяти между этими двумя элементами одного массива, причем расстояние выражено не в байтах, а в типе short, который занимает 2 байта. Фактически, значение res вычисляется по формуле:

```
res = (6487814 – 6487808) / 2 = 3
```

Здесь деление на 2 – это, как раз, следствие адресной арифметики (тип short занимает 2 байта).

Еще раз обратите внимание, что операция разности между двумя указателями следует выполнять только с элементами одного и того же массива. Во всех остальных случаях получаем неопределенное поведение.

Пример использования адресной арифметики

Вот вам и вся адресная арифметика. А чтобы было понятнее, приведу один показательный пример ее использования.

Пусть в программе по-прежнему объявляется целочисленная переменная типа int и ставится задача просмотреть побайтно содержимое этой переменной в памяти компьютера. Сделать это можно следующим образом:

```
#include <stdio.h>

int main(void)
{
        int g = 476789;
        char *ptr = (char *)&g;

        for(int i = 0;i < sizeof(g); ++i) {
                printf("%d ", *ptr);
                ptr++;
        }

        return 0;
}
```

Смотрите, мы здесь формируем указатель, который работает с байтовыми данными, то есть, с отдельными ячейками памяти. Затем, ему присваивается адрес целочисленной переменной g и в результате он ссылается на первый байт этой переменной. После этого в цикле for осуществляется вывод текущего значения байта на экран и указатель ptr увеличивается на единицу. Так как тип у него прописан как char, то операция инкремента увеличит адрес ptr ровно на один и мы перейдем к следующему байту. Соответственно, на следующей итерации будет выведено значение очередного байта и так для всех ячеек переменной int. В итоге на экране увидим числа:

```
117 70 7 0
```

которые в точности определяют число:

$$476789 = 117 + 256 \cdot 70 + 256^2 \cdot 7$$

Приоритеты операций при работе с указателем

Кстати, тело цикла в программе можно было бы записать и короче:

```
for(int i = 0;i < sizeof(g); ++i)
        printf("%d ", *ptr++);
```

Здесь у нас две унарные операции ++ и * применяются к указателю ptr. Спрашивается, в каком порядке будут происходить вычисления? Здесь следует вспомнить, что приоритет унарных операций убывает справа-налево. Поэтому сначала идет инкремент в постфиксной форме и только затем операция разыменования. Это эквивалентно такой записи:

```
*(ptr++)
```

Так как инкремент записан в постфиксной форме, то вначале мы получаем текущее значение ptr, к нему применяется операция * и только после этого адрес увеличивается на единицу.

А вот если инкремент записать в префиксной форме:

```
for(int i = 0;i < sizeof(g); ++i)
        printf("%d ", *++ptr);
```

то это будет эквивалентно записи:

```
*(++ptr)
```

Здесь сначала адрес указателя увеличивается на единицу и только после этого срабатывает следующая операция разыменования. Поэтому байты будут прочитаны со сдвигом вправо на одну ячейку:

```
70 7 0 3
```

Вообще, когда мы используем указатель совместно с операцией разыменования и адресной арифметикой, то следует хорошо знать и учитывать их приоритеты. Например, имеется следующий фрагмент программы:

```
int g = 476789;
int *p = &g;

*p += 1;
```

Спрашивается, как будет работать последняя строчка? Рассуждаем здесь подобным образом. Так как операция * является унарной, то она обладает большим приоритетом, чем операция +=. Поэтому здесь сначала будет прочитано значение переменной g, затем, оно увеличивается на единицу и результат снова заносится в те же ячейки памяти, где расположена переменная g. В итоге, значение указателя p не изменится, а переменная g станет на единицу больше.

Если вы не уверены в порядке выполнения операций, то вполне допустимо использовать круглые скобки. Например, имеется программа:

```
#include <stdio.h>

int main(void)
{
        int g = 476789;
        char *ptr = (char *)&g;

        int x = *ptr + 1;
        printf("x = %d\n", x);

        return 0;
}
```

И спрашивается, чему будет равно значение переменной x? То есть, как отработает операция *ptr + 1? Очевидно, здесь приоритет унарной операции * выше, чем у бинарной операции сложения. Поэтому, сначала будет прочитано значение из первого байта переменной g – это число 117, а затем, к нему будет прибавлена единица. В итоге x будет содержать число 118.

А вот если эту же строчку записать с круглыми скобками следующим образом:

```
int x = *(ptr + 1);
```

то ситуация кардинально меняется. Сначала будет увеличен адрес на единицу, мы перейдем к следующей ячейке переменной g, и переменной x будет присвоено значение этой второй ячейки. В итоге она будет принимать значение 70.

И раз еще давайте посмотрим на работу команды:

```
int x = *ptr++;
```

Такая запись указателя с операцией инкремента и разыменованием часто используется в практике программирования. В итоге мы здесь читаем значение из текущей ячейки, и только после это адрес указателя увеличивается на единицу. Но, если мы это же выражение запишем в виде:

```
int x = (*ptr)++;
```

то операция инкремента будет применена уже к данным в первой ячейке переменной g. В итоге переменной x присвоится начальное значение из первой ячейки, а переменная g станет содержать число:

$$476790 = 118 + 256 \cdot 70 + 256^2 \cdot 7$$

Как видите, все работает достаточно просто и логично. Возможно, если вы только делаете первые шаги в этой теме, нужно немного привыкнуть к особенностям работы указателей. Но ровным счетом ничего сложного в понимании их работы нет. Очень скоро, при определенной практике, каждый из вас сможет грамотно применять их в своих программах.

Практический курс по C/C++: https://stepik.org/course/193691

Видео по теме


Язык Си. Рождение легенды


#1. Этапы трансляции программы в машинный код. Стандарты


#2. Установка компилятора gcc и Visual Studio Code на ОС Windows


#3. Структура и понимание работы программы "Hello, World!"


#4. Двоичная, шестнадцатеричная и восьмеричная системы


#5. Переменн базовые т Модификаторы

← Предыдущая