Задание практикума №1

3 августа 2022 г.

К исследованию было предложено 2 задачи. В первой нам необходимо проверить существование «ленивых вычислений» в языке Си. Вторая предлагает нам определить каким образом при выполнении операции присваивания и явном приведении происходит преобразование вещественных чисел (X) к беззнаковым целым (N-битовое представление).

1 Задание 10 - ленивые вычисления

Ленивые вычисления – это специальная стратегия вычислений, позволяющая откладывать некоторые вычисления до тех пор, пока не понадобятся их результаты. Иными словами, мы можем игнорировать некоторые выражения, если заранее знаем, что они не повлияют на конечный результат. Проверим, действительно ли это так:

```
1  #include <stdio.h>
2
3  int check_1()
4  {
5    printf("Well, we are wrong - part 1");
6    return 0;
7  }
8
9  int check_2()
10 {
11   printf("Well, we are wrong - part 2");
12   return 1;
13 }
14
15 int
16 main(void)
17 {
18   printf("Result 1: %d", 1 || check_1);
19   printf("\n");
20   printf("Result 2: %d", 0 && check_2);
21   return 0;
22 }
```

В строках 18 и 20 мы хотим вывести результаты логических выражений. Частями этих составных высказываний являются функции. Если мы

не правы и ленивые вычисления не используются языком Си, то функции будут запущены и маркер выведет нам на экран дополнительное сообщение:

```
Result 1: 1
Result 2: 0
```

Как видно из вывода, маркер не напечатан, значит, функции не были запущены. Отсюда делаем вывод, что язык Си использует ленивые вычисления. Но в чем смысл такой стратегии вычислений? Предполагается, что раз часть выражений не считается, то результат может быть получен за меньшее время. Проверим теорию на практике:

```
#include <stdio.h>
#include <time.h>
int
main(void)
{
  int i, k, t;
  for (i = 0; i < 100000000; i++)
  {
    k = 0 || 1;
  }
  t = clock();
  printf("Time in long process = %f", ((float)t) / CLOCKS_PER_SEC);
  return 0;
}
```

Затраченное время:

```
Time in long process = 0.152000
```

В данном примере не могут быть использованны ленивые вычисления, поэтому приходится искать результаты всех выражений. Теперь тот же пример, но уже с возможностью использования хитрой стратегии:

```
#include <stdio.h>
#include <time.h>
int
main(void)
{
  int i, k, t;
  for (i = 0; i < 100000000; i++)
  {
    k = 1 || 0;
  }
  t = clock();
  printf("Time in short process = %f", ((float)t) / CLOCKS_PER_SEC)
    ;
  return 0;
}
```

Затраченное время:

```
Time in short process = 0.146000
```

Легко заметить, что время во втором примере меньше времени в первом, а значит ленивые вычисления позволяют получить результат общего выражения быстрее за счет игнорирования отдельных его частей, не влияющих на итоговый ответ.

Таким образом, мы доказали, что ленивая стратегия вычислений используется языком Си.

2 Задание 19 - преобразование вещественного числа в беззнаковое целое

Преобразования типов бывают двух видов: явные и неявные. Неявные выполняются компилятором в соответствии со стандартами языка. Для явных же преобразований используются функции, принимающие на вход один тип и возвращающие другой, а также специальные конструкции самого языка, доступные программисту.

Рассмотрим конкретный пример - преобразование вещественного типа double к целому беззнаковому unsigned long long int. Проверим 3 случая: число больше нуля, число меньше нуля, число превышает допустимые размеры типа. Ниже приведен код, выводящий двоичное представление вещественного числа, а так же выводящий его двоичный и десятичный вид после преобразования в целый беззнаковый тип.

```
#include <stdio.h>
union bi
{
    double b;
    unsigned long long int a;
};

int bitprinter_1(union bi K)
{
    printf("\n");
    printf("%lf", K.b);
    printf("\n");
    unsigned long long int mask = (unsigned long long int)1 << 63;
    while (mask != 0)
    {
        printf("%d", (K.a & mask) == 0 ? 0 : 1);
        mask = mask >> 1;
    }
    printf("\n");
    return 0;
}

int bitprinter_2(unsigned long long int a)
{
    printf("\n");
    unsigned long long int mask = (unsigned long long int)1 << 63;
    while (mask != 0)
    {
        printf("%d", (a & mask) == 0 ? 0 : 1);
        mask = mask >> 1;
    }
    printf("\n");
    printf("%llu", a);
    printf("\n");
```

```
    return 0;
}

int
main(void)
{
    union bi test_1, test_2, test_3;
    unsigned long long int t_1, t_2;
    unsigned short int t_3;

    test_1.b = 345.035;
    printf("Firts test 0 <= X < 2^N \n");
    printf("Before: ");
    bitprinter_1(test_1);
    t_1 = (unsigned long long int)test_1.b;
    printf("After: ");
    bitprinter_2(t_1);
    printf("\n");
    test_2.b = -250.0;
    printf("Second test X < 0 \n");
    printf("Before: ");
    bitprinter_1(test_2);
    t_2 = (unsigned long long int)test_2.b;
    printf("After: ");
    bitprinter_2(t_2);
    printf("\n");
    test_3.b = 72000.45;
    printf("Third test X >= 2^N \n");
    printf("Before: ");
    bitprinter_1(test_3);
    t_3 = (unsigned short int)test_3.b;
    printf("After: ");
    bitprinter_2(t_3);
    return 0;
}
```

Посмотрим на вывод:

```
Firts test 0 <= X < 2^N
Before:
345.035000
01000000011101011001000010001111010111000010100011111010111000011
After:
00000000000000000000000000000000000000000000000000000000101011001
345

Second test X < 0
Before:
-250.000000
11000000011011110100000000000000000000000000000000000000000000000
After:
11111111111111111111111111111111111111111111111111111111100000110
18446744073709551366

Third test X >= 2^N
Before:
72000.450000
01000000111100011001010000000111001100110011001100110011001100110011
```

```
After:
00000000000000000000000000000000000000000000000001100101000000
6464
```

В первом случае, когда вещественное число больше нуля, но помещается в размер типа, преобразование происходит посредством отбразывания дробной части. Большая разница в двоичном представлении этих чисел наблюдается из-за принципов представления вещественных и целых чисел в памяти компьютера.

Во втором случае если вещественное число было меньше нуля, алгоритм меняется. Дробная часть также отбрасывается. Однако знак "минус"машина игнорировать не может. Тогда в переменную беззнакового целого типа закладывается дополнительное к нашему число. Дополнительное число получается посредством сложения отрицательного числа и 2 в степени 64(т.к. число в нашем типе представляется 64 битами). Происходит закольцованность типа. И действительно, если сложить 250 и полученное число, получится 2 в степени 64 - таким образом, кольцо замыкается.

Третий случай, когда число выходит за размеры типа, стоит рассматривать очень внимательно. Как и прежде, дробная часть отбразывается. Но что происходит с осташейся целой частью числа? Ведь она сама по себе будет больше преобразуемого типа. В такой ситуации машина принимает решение отбрасывать старшие биты до тех пор, пока число не влезет в нужный тип.

Что же, мы смогли посмотреть, как выглядят представления чисел при явных преобразованиях. Но довольно интересно будет взглянуть на то, что происходит на машинном уровне. Вот код с явным преобразованием на языке Си:

```c
#include <stdio.h>
int
main(void)
{
  float a;
  unsigned int b = (unsigned int)a;
  return 0;
}
```

А вот ассемблеровская часть кода, отвечающая за строку 6:

```
;unsigned int b = (unsigned int)a;
  fld DWORD PTR 28[esp]
  fnstcw  WORD PTR 14[esp]
  movzx eax, WORD PTR 14[esp]
  or  ah, 12
  mov WORD PTR 12[esp], ax
  fldcw WORD PTR 12[esp]
  fistp QWORD PTR [esp]
  fldcw WORD PTR 14[esp]
  mov eax, DWORD PTR [esp]
```

```
11  mov edx, DWORD PTR 4[esp]
12  mov DWORD PTR 24[esp], eax
```

Команды вида -f... являются командами сопроцессора. Команды fnstcw и fldcw являются управляющими командами. Они предназначены для работы с нечисловыми регистрами сопроцессора.

Теперь разберем подробно, за что отвечает каждая строка кода:
1) Команда fld закружает в верщину стека вещественное число.
2) Команда fnstcw записывает содержимое управляющего регистра в оперативную память.
3) Команда movzx перемещает(с расширением) в регистр eax содержимое управляющего регистра, находящееся в оперативной памяти.
4) В биты 2 и 3 регистa eax закладывается единица - запрещаются прерывания в случае ошибки деления на 0(маска zm) и переполнения(маска om) соответственно.
5) В оперативную память мы закладываем значение регистра ax - измененное значение управляющего регистра.
6) Команда fldcw загружает управляющий регистр данными из оперативной памяти - регистра ax (т.е. обновляется значение управляющего регистра).
7) Команда fistp выполняет извлечение из стека в память целого числа.
8) Команда fldcw снова загружает данные из памяти(содержимое управляющего регистра до изменения его битов в строке 4).
9-10) Целое число из памяти записывается на регистры eax и edx(т.к. извлекли мы 8 байт в строке 7).
11) Приведенное целое число записывается из регистра eax обратно в память(загружаем только 4 байта, т.к. размер беззнакового целого равен 4 байтам и в регистре edx число не лежит).

Таким образом, мы рассмотрели алгоритмы явного приведения вещественного числа к беззнаковому целлому в трех случаях, а так же посмотрели, что происходит на машинном уровне при такой задаче.