

Увод в програмирането

Лекция 4: **Цикли (втора част)**

Преговор

- for, while, do-while
- +=, -=, ..., ++, --
- Област на променлива
- Задача: Да се намери сумата на квадратите на всички четни числа в $[a, b]$ (a, b - цели)

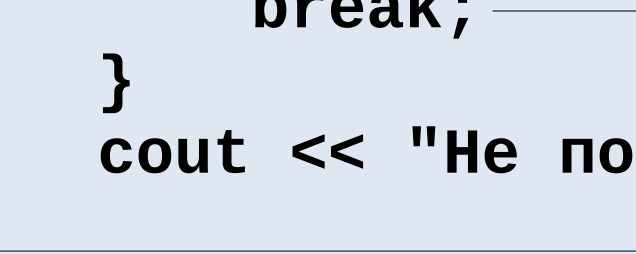


Прекратяване на цикъл

- Цел: искаме в даден момент да прекратим изпълнението на оператор за цикъл, независимо, че му остават още итерации
- Ще разгледаме два начина:
 - С използването на нов оператор – `break`
 - С използване на познати средства – булев флаг

Прекратяване на цикъл с break

- Оператор **break**; (познат от switch)
- ```
cout << "Познай от 3 пъти колко имам по УП: ";
for (int i = 0; i < 3; i++) {
 int grade;
 cin >> grade;
 if (grade == 6) {
 cout << "Точно така, позна!" << endl;
 break;
 }
 cout << "Не позна :(" << endl;
}
```


- break се слага в рамките на някакъв условен оператор, в противен случай е безсмислено – ще се спре още на първата итерация

# Прекратяване на цикъл с флаг

- Използване на *флаг*
- Според някои източници използването на `break` не е добра практика, според други е
- ```
cout << "Познай от 3 пъти колко имам по УП: ";  
bool correct = false; // флаг  
for (int i = 0; i < 3 && !correct; i++) {  
    int grade;  
    cin >> grade;  
    if (grade == 6) {  
        cout << "Точно така, позна!" << endl;  
        correct = true;  
    } else { // преди нямаше нужда от else  
        cout << "Не позна :(" << endl;  
    }  
}
```

Пример

- Да се въведат цели положителни числа и да се намери сумата им.
Въвеждането да спре при срещане на 0 или на отрицателно число (и то да не участва в изчислението на сумата).
- ```
int sum = 0, number;
cin >> number;
while (number > 0) {
 sum += number;
 cin >> number;
}
cout << sum << endl;
```
- ```
// 2-и начин:  
int sum = 0, number;  
while (true) { // безкраен  
                // цикъл  
    cin >> number;  
    if (number <= 0)  
        break;  
    sum += number;  
}
```

Прекратяване на вложени цикли

- break прекратява само най-вътрешния цикъл

- **bool flag = true;**

```
for (int i = 0; i < 10 && flag; i++) {  
    for (int j = 0; j < 10; j++) {  
        cout << i << ", " << j << endl;  
        if (i + j > 10) {  
            flag = false;  
            break;  
        }  
    }  
}
```

- Има ли значение в какъв ред са операторите `flag = false;` и `break;` ?

Прекратяване на итерация

- Оператор **continue**;
- За разлика от `break`, `continue` прекратява само изпълнението на текущата итерация
- Продължава се със следващата итерация
- Ако цикълът е `for`, след `continue` се изпълнява третият компонент на `for`
- Отново няма смисъл да се слага `continue`, ако ще бъде изпълнен безусловно
 - Кодът след `continue` няма да се изпълни в нито една итерация

Пример за continue

- Средно аритметично на всички числа от 0 до $n - 1$, които не се делят нито на 3, нито на 7:

```
■ int sum = 0, count = 0, n;  
  cin >> n;  
  for (int i = 0; i < n; i++) {  
      if (i % 3 == 0 || i % 7 == 0)  
          continue;  
      sum += i;  
      count++;  
  }  
  cout << (count ? sum / count : 0);
```

- Кой забрави проверката дали `count != 0`? :)

Няма нужда да ползваме continue!

- Еквивалентно решение без continue:
- ```
int sum = 0, count = 0, n;
cin >> n;
for (int i = 0; i < n; i++) {
 if (!(i % 3 == 0 || i % 7 == 0)) {
 sum += i;
 count++;
 }
}
cout << (count ? sum / count : 0);
```
- Условието в конкретния пример може да се опрости с едно от правилата на Де Морган

**Задачи за съществуване и „за всяко“**

# Шаблонни задачи

- Много на пръв поглед различни проблеми се решават с еднотипни алгоритми:
  - Дали цяло число съдържа цифрата 5
  - Дали цяло число е просто
  - Дали всички цифри на едно число са еднакви
  - Дали цифрите на едно число образуват нарастваща редица
  - Дали цифрите на число образуват палиндром
  - И още много, много други

# Задача за съществуване (1)

- Да се провери дали в дадено множество  $A$  се съдържа елемент  $x$ , който удовлетворява дадено свойство  $p(x)$ 
  - Например: дали число съдържа цифрата 5:
    - $A$  – множество от цифрите на това число
    - $x$  – цифра
    - $p(x)$  – дали  $x == 5$

# Задача за съществуване (2)

- Псевдокод:
- ```
bool found = false;  
<цикъл> // обхождаме множеството  
{  
    if (текущият елемент удовлетворява търсеното свойство)  
    {  
        found = true;  
        break; // или в условието на цикъла има && !found  
    } // няма else!  
}
```
- Ако не съществува търсеният елемент, found си остава false

Задача за съществуване (3)

- Ако съществува елемент, удовлетворяващ даденото свойство, цикълът се прекратява
 - Това не е задължително, но ускорява изпълнението на програмата

Пример

- Програма, която проверява дали десетичният запис на цяло число n съдържа цифрата k

Задачи „за всяко“

- Да се провери дали в дадено множество A всеки елемент x удовлетворява дадено свойство $p(x)$
 - Например: дали цяло число е просто – дали всяко число, по-малко от него (или е по-малко или равно на неговия квадратен корен) не дели даденото число
 - $A = [2, \dots, \lfloor \sqrt{n} \rfloor]$
 - $p(x) = n \% x \neq 0$

Алгоритъм за решаване на задача за всяко

- Да разгледаме пример: искаме да проверим дали всички ябълки са здрави
- Какво значи всички ябълки да са здрави?
- Да не съществува гнила ябълка измежду тях
- Т.е. ако ги обхождаме една по една и намерим гнила ябълка, спираме и обявяваме резултат – false, не е вярно, че всички са здрави
- Ако не намерим – true, всички са здрави



Свеждане до задача за съществуване

- От примера се вижда, че проверката дали всички елементи на едно множество удовлетворяват дадено свойство е всъщност задача за търсене на елемент, НЕудовлетворяващ свойството
- Но при намиране на елемент обявяваме резултат false, а не true, и обратното
- $\forall x \in A: p(x) \Leftrightarrow \neg \exists x \in A: \neg p(x)$
- $\exists x \in A: p(x) \Leftrightarrow \neg (\forall x \in A: \neg p(x))$ (има гнила ябълка – значи не е вярно, че всички са здрави)

- Псевдокод:
- `bool flag = true;`
`<цикъл> // обхождаме множеството`
`{`
 `if (текущият елемент НЕ удовлетворява даденото свойство)`
 `{`
 `flag = false;`
 `break; // или в условието на цикъла има && flag`
 `} // няма else!`
`}`
- Тоест решаваме задача за съществуване, но проверяваме обратното свойство и обръщаме резултата – при намиране казваме false

- Предложеният алгоритъм отново е ефективен при намиране на елемент, неудовлетворяващ свойството – цикълът се прекратява
- По-лош, но все пак коректен алгоритъм: да броим елементите, които удовлетворяват условието и накрая да сравним този брой с броя на всички елементи
 - Ако още първият елемент не е „хубав“, защо да губим време да проверяваме и следващите?
 - Посоченият по-лош алгоритъм често се среща сред студентските решения

Примери

- Вж. началния слайд на секцията
- По-сложни задачи: вложени проверки за съществуване/всяко:
 - Дали съществува просто число в даден интервал
 - Дали цифрите на едно число са различни

По-труден и допълнителен материал

Пресмятане на конюнкция и дизюнкция (1)

- Нека имаме $A \ \&\& \ B$
- Изчислява се стойността на A и тя се оказва `false`
- Има ли смисъл да се изчислява B ?
 - `false && false` \rightarrow `false`
 - `false && true` \rightarrow `false`
- Няма смисъл, затова компилаторът генерира такъв машинен код, който не изчислява B в такава ситуация
- Естествено, ако A е `true`, B също се смята

Пресмятане на конюнкция и дизюнкция (2)

- Аналогично при $A \parallel B$:
- Ако A има стойност `true`,
 B не се изчислява –
независимо от стойността му, цялата
дизюнкция има стойност `true`

Пресмятане на конюнкция и дизюнкция (3)

- Тази оптимизация как влияе на нашите програми?
 - Ускорява ги – ако дясната страна е бавна, например изтегля файл от интернет и проверява съдържанието му
 - Ако в дясната страна имаме операции със страничен ефект (напр. +=), или такива, които предизвикват грешка (напр. деление на 0), резултатът от изпълнението на програмата е различен спрямо резултата при липса на такава оптимизация

Примери (подходящи и за теоретичния изпит)

- ```
bool b = false;
int c = 5;
if (b && c++)
 c = 4324589;
cout << c << endl; // Колко е?
if (c++ && b)
 c = -17;
cout << c << endl; // А тук?
```

# Пример

- Искаме да проверим дали частното на целите числа  $a$  и  $b$  е четно число
  - Щом имаме деление, нашата програма трябва да се държи коректно дори и ако знаменателят е 0 (а не да "гърми")
- `if (b != 0 && a / b % 2 == 0)`  
    `cout << "Yes";`

# Много труден пример

- Какво ще се отпечата и защо?
- ```
const int size = 4;  
for (int i = 2, k = 6; --k; i -= i || (i += size))  
    cout << i + 1;
```

Отново за for и while (1)

- Почти винаги
for (A; B; C) D
е еквивалентно на
{A; while (B) {D; C;}}
- Външните големи скоби може да ги сложим, ако не искаме променливите, дефинирани в A, да се виждат след края на цикъла
- Кога двете конструкции не са еквивалентни?

Отново за for и while (2)

- ```
for (int i = 0; i < 4; i++) {
 if (i == 2) continue;
 cout << i << " ";
}
```

- Ще се отпечата: 0 1 3

- ```
{  
    int i = 0;  
    while (i < 4) {  
        if (i == 2) continue;  
        cout << i << " ";  
        i++;  
    }  
}
```

- Отпечатват се 0 1 и програмата зацикля

Допълнителен материал:

Comma separated expressions

- Ако $\langle \text{израз}_1 \rangle$ и $\langle \text{израз}_2 \rangle$ са два израза, то $\langle \text{израз}_1 \rangle, \langle \text{израз}_2 \rangle$ също е израз
- При изчисляването му се изчисляват последователно левият и десният израз
- Стойността на новия израз е стойността на последния израз
- Примери:
- `cout << (1, 2);`
`for (int i = 0, j = 0; i < 5 && j < 5; i++, j++) ;`

Допълнителен материал

- Системи за контрол на версиите – git

Обобщение и въпроси