Pętle

Pętla to pewien fragment kodu, który jest wykonywany wielokrotnie. Wyobraź sobie taką sytuację. Piszesz program do szyfrowania danych. Dane są szyfrowane kolejno bajt po bajcie. Załóżmy, że plik zawiera 800 bajtów. Jak to zrobić? Napisanie 800 razy tej samej instrukcji szyfrującej oczywiście odpada. Po za tym co w sytuacji, gdy plik ma 900 bajtów. Wtedy trzeba byłoby ingerować w kod programu. Sam widzisz, że takie rozwiązanie nie wchodzi w grę. Należy wykonać szyfrowanie jakoś sprytniej :) Czy nie fajnie byłoby napisać instrukcję szyfrującą jednokrotnie i wykonać odpowiednią ilość razy? Otóż można! Do takich przypadków służą pętle.

PĘTLA WHILE

Jak zawsze wyjdziemy od terminologii. While oznacza *dopóki, podczas gdy*. Pętla `while` jest wykonywana **dopóki** postawiony warunek jest spełniony. Taka pętla może zawierać jedną lub więcej instrukcji.

```
1. while (5 > 3) instrukcja;
```

Widzisz podobieństwo do instrukcji `if`? Taka pętla raczej nie znajdzie zastosowania w praktyce. Warunek jest zawsze ten sam. Zawsze jest on prawdziwy. Oznacza to, że pętla, a tym samym będąca w pętli instrukcja będzie wykonywana w nieskończoność. Oczywiście nic nie stoi na przeszkodzie, aby w pętli użyć zmiennej, jako warunku:

```
1. while (zmienna != 0) instrukcja;
```

Tutaj instrukcja będzie wykonywana **dopóki** wartość zmiennej będzie różna od zera. Jeżeli pętla zawiera kilka instrukcji istnieje możliwość umieszczenia ich w bloku. Robi się to analogicznie jak przypadku instrukcji `if`:

```
1. while (zmienna < 100)
2. {
3. instrukcja_1;
4. instrukcja_2;
5. instrukcja_3;
6. }
```

Podczas każdorazowego obiegu pętli nastąpi wykonanie wszystkich trzech instrukcji. Oczywiście pętla będzie się wykonywać dopóki zmienna będzie mniejsza od stu. W przeciwnym wypadku nastąpi wyskok z pętli. Należy również pamiętać, że warunek jest tutaj sprawdzany w momencie wchodzenia do pętli. Oznacza to, że jeśli na początku nie zostanie on spełniony to pętla **nie wykona się nawet jeden raz!**

PĘTLA DO-WHILE

Właściwie jest to jakby rozszerzenie pętli `while` i nie warto sobie tym zawracać głowy. Jednak dla formalności powiem kilka słów. Zasadnicza różnica pomiędzy tą pętlą, a wcześniejszą jest taka, że tutaj nastąpi **przynajmniej jeden obieg**. Pierwszy obieg jest niezależny od warunku, ponieważ jego sprawdzenie nastąpi **dopiero** na końcu. Tutaj zostanie podjęta decyzja dotycząca ponownego obiegu. Składniowo przedstawia się to tak:

```
1. do
2. {
3. instrukcja_1;
4. instrukcja_2;
5. instrukcja_3;
6. }
7. while (zmienna > 10);
```

Po za wspomnianą różnicą reszta jest niezmienna. Czyli teraz wszystkie instrukcje zostaną wykonane **przynajmniej** jednokrotnie. Jeżeli przy kończeniu zmienna będzie większa od dziesięciu nastąpi kolejny obieg. I tak w kółko. Pamiętaj tylko o średniku na końcu! Pętle `do-while` i `while` stosowane są najczęściej w sytuacjach, gdy nieznana jest ilość obiegów. Podaje się warunek i pętla jest wykonywana lub nie. Czasami takie rozwiązanie się przydaje. Jednak przeważnie wiemy ile razy dana pętla zostanie wykonana. Wtedy można posłużyć sie inną pętla.

PĘTLA FOR

Pętla `for` jest wykorzystywana w sytuacjach, gdy można określić bliżej ile razy będzie ona powtarzana. Słowo `for` oznacza *dla*. I tak dla przykładu:

```
1. for (wyrażenie_początek; warunek; instrukcja_co_obieg)
2. {
3. instrukcja_1;
4. instrukcja_2;
5. instrukcja_3;
6. }
```

Wyraźnie widać, że już w składni pojawiają się znaczne różnice. Działanie ogólnie to samo. Wykonanie określonej czynności wiele razy. W nawiasie idąc od lewej znajduje się instrukcja, która jest wykonywana na początku **pierwszego** obiegu. Podczas każdego następnego sprawdzany jest warunek. Na tej podstawie stwierdza się, czy kontynuować wykonywanie czy nie. Następnie jest dowolna instrukcja, która jest wykonywana co obieg pętli. Przeważnie pętla `for` wygląda następująco:

```
1. for (int i = 0; i < 100; i++)
2. {
3. instrukcja_1;
4. instrukcja_2;
5. instrukcja_3;
6. }
```

Pętla `for` w takiej postaci działa w taki sposób. Najpierw jest definiowana zmienna o nazwie **i**. Dodatkowo jest ona inicjalizowana wartością zero. Przy każdym obiegu pętli następuje inkrementacja licznika **i**. Gdy osiągnie on wartość 99 pętla zakończy się. Zauważ, że pomimo liczby 99 pętla zostanie wykonana 100 razy. To dlatego, że rozpoczynamy od zera. Tutaj również klamry nie są obowiązkowe jeśli pętla zawiera **tylko** jedną instrukcję. Mało tego. Podczas pisania pętli `for` wcale nie musisz podawać wszystkich informacji. Możesz pominąć warunek wyskoku z pętli, tak jak poniżej:

```
1. for (int i = 0; ; i++)
2. {
3. instrukcja_1;
4. instrukcja_2;
```

```
5. instrukcja_3;
6. }
```

Jest to prawie taka sama pętla jak poprzednio. Zauważ, że **nie** ma w niej warunku, który jest sprawdzany przed każdym obiegiem. W takiej sytuacji pętla będzie wykonywana bez przerwy. Pamiętaj, że jeśli pomijasz jakąś informację znajdującą się w nawiasie **musisz** pozostawić średnik! Istnieje też możliwość umieszczenia kilku informacji w części inicjalizacyjnej. Wówczas wtedy należy je oddzielić przecinkiem. W kodzie przedstawia się to w następujący sposób:

```
1. for (int i = 0, char c = 10, short s = 100; ;i++) instrukcja;
```

Jednak takie kombinacje zdarzają się rzadko. Najczęściej korzysta się głównie z jednego sposobu. **Definicja licznika, ilość wykonań, inkrementacja licznika**. Zresztą taki przykład już jest kilka linijek wyżej.

DODATKOWE INSTRUKCJE W PĘTLACH

Jak dotąd rozważaliśmy przypadki, gdy pętla wykonywana była zależnie od warunku. W pętli `while` i `do` znajdował się on na początku lub na końcu. W pętli `for` także był sprawdzany na początku. Teraz jak można nieco *nagiąć* takie przepisy i wyskoczyć z pętli w dowolnej chwili.

BREAK

Instrukcję `break` już poznałeś przy okazji omawiania instrukcji `switch`. Tam służyła ona do wyskoku z bloku `switch`. W pętlach jej działanie jest identyczne. Za jej pomocą można szybko wyskoczyć z pętli.

```
1. for (int i = 0; i < 30; i++)
2. {
3. instrukcja_1;
4. instrukcja_2;
5. instrukcja_3;
6. if (zmienna == 7.78) break; //ooo sorr'y tutaj wyskakuję :)
7. }
```

Normalnie pętla wykonywałaby się 30 razy. Jednak wartość zmiennej to jakaś szczególna liczba. Dlatego też jeżeli będzie ona miała wartość 7.78 wyskoczymy. Jak pamiętasz mówiłem, że instrukcję

if mogą znajdować się wewnątrz innej instrukcji **if**. Mówi się wtedy o tzw. zagnieżdżeniu instrukcji **if** . W przypadku pętli również jest to dozwolone. Jest to praktyka dość często stosowana. Wygląda to tak:

```
1. for (int x = 0; x < 30; x++)
2. {
3. for (int y = 0; y < 30; y++) //pętla druga - zagnieżdżona
4. {
5. instrukcja_1;
6. instrukcja_2;
7. if (zmienna == 7.78) break; //ooo sorr'y tutaj wyskakuję :)
8. } //zakończenie pętli drugiej
9. } //zakończenie pętli pierwszej
```

Zauważ, że instrukcja **break** została umieszczona w pętli zagnieżdżonej. Jeżeli nawet zmienna będzie równa 7.78 to wyskok nastąpi o jeden poziom wyżej. Oznacza to, że będziemy w pętli `for`, tylko tej *zewnętrznej*. Jednak nie o taki wyskok tutaj chodziło. Aby wyskoczyć *na zewnątrz* trzeba w **każdej** pętli umieścić instrukcję wyskoku. Inaczej się nie da.

CONTINUE

Instrukcję **continue** jest również stosowana w odniesieniu do pętli. Służy do pominięcia aktualnego obiegu i rozpoczęcia następnego. Jest dość rzadko stosowana. Jednak czasem przydaje się.

```
1. for (int i = 0; i < 30; i++)
2. {
3. if (zmienna == 7.78) continue; //ooo sorr'y tutaj wyskakuję :)
4. instrukcja_1;
5. instrukcja_2;
6. }
```

Tym razem jeżeli zmienna wyniesie 7.78 bieżący obieg pętli zostanie zaniechany i rozpocznie się obieg następny. Instrukcja **continue** została umieszczona na początku, dlatego też **wszystko** co znajduje się za nią **nie** zostanie wykonane w tym obiegu.