Wstęp do informatyki- wykład 6

Operatory – przypisania, złożone operatory przypisania, operator przecinkowy

Pętla `while` i `do..while`

Treści prezentowane w wykładzie zostały oparte o:

- S. Prata, Język C++. Szkoła programowania. Wydanie VI, Helion, 2012
- *www.cplusplus.com*
- Jerzy Grębosz, Opus magnum C++11, Helion, 2017
- B. Stroustrup, Język C++. Kompendium wiedzy. Wydanie IV, Helion, 2014
- S. B. Lippman, J. Lajoie, Podstawy języka C++, WNT, Warszawa 2003.

Operatory inkrementacji i dekrementacji *(grupa 2)*

Operatory inkrementacji (zwiększenia o jeden) i dekrementacji (zmniejszenia o jeden).
Zwiększenie o 1 lub zmniejszenie o 1 jest działaniem tak często spotykanym w programowaniu, że w języku C++ mamy specjalne operatory:

```
i++ ; // czyli to samo co: i = i + 1
k-- ; // czyli to samo co: k = k - 1
```

Operatory inkrementacji ++ i dekrementacji -- są jednoargumentowe/ Wiązanie mają prawe. Oba mogą mieć dwie formy:

- przedrostkowa (prefiks) – wtedy, gdy operator stoi z lewej strony argumentu, np. ++a, `--p` (czyli przed argumentem);
- końcówkowa (postfiks) – wtedy, gdy operator stoi po prawej stronie argumentu, np. a++, `p--` (po prostu po argumencie).

Operatory inkrementacji i dekrementacji *(grupa 2)*

Jest między nimi pewna różnica. Rozważmy to na przykładzie operatora inkrementacji (zwiększania).

- Jeśli operator inkrementacji stoi przed zmienną (np.`++i`), to:
    - najpierw jest ona zwiększana o 1,
    - następnie ta zwiększona już wartość staje się wartością wyrażenia.
- W przypadku gdy operator inkrementacji stoi za zmienną (np.`i++`), to:
    - najpierw brana jest stara wartość tej zmiennej i ona staje się wartością wyrażenia,
    - a następnie dopiero wartość zwiększana jest on o 1. Zwiększenie to nie wpłynęło więc jeszcze na wartość samego wyrażenia.

Operatory inkrementacji i dekrementacji *(grupa 2)*

```
int main()
{
   int a = 5, b = 5, c = 5, d = 5;
   cout<<"Wstepnie\n a = " << a << ", b = " << b
       << ", c = " << c << ", d = " << d << endl;
   cout<< "Wartosc poszczegolnych wyrazen\n";
   cout<< "++a = " << ++a << ", b++ = " << b++
       <<", --c = " << --c << ", d-- = " << d--
       <<endl;
    // teraz sprawdzamy, co jest obecnie w zmiennych
   cout<<"Wartosci zmiennych po obliczeniu wyrazen\n"
       << " a = " << a << ", b = " << b
       << ", c = " << c << ", d = " << d << endl;
   return 0;
}
```

Operatory inkrementacji i dekrementacji *(grupa 2)*

W rezultacie otrzymujemy:

```
Wstepnie
 a = 5, b = 5, c = 5, d = 5
Wartosc poszczegolnych wyrazen
++a = 6, b++ = 5, --c = 4, d-- = 5
Wartosci zmiennych po obliczeniu wyrazen
 a = 6, b = 6, c = 4, d = 4
```

Operator inkrementacji stojący przed argumentem nazywa się często **operatorem preinkrementacji**,

natomiast stojący za argumentem nazywa się **operatorem postinkrementacji**.

Podobnie jest w przypadku operatorów zmniejszania – mówimy o operatorach **predekrementacji** i **postdekrementacji**.

Instrukcja wyboru `switch`

Instrukcja `switch` służy do podejmowania wielowariantowych decyzji. W zasadzie zawsze może być zastąpiona instrukcjami `if`, ale czasem czytelniej jest użyć właśnie instrukcji `switch`.

Jej najbardziej ogólna postać to:

```
switch (wyr_calk)
{
  case stala1: lista1
  case stala2: lista2
  // ...
  default: lista
}
```

gdzie **`wyr_calk`** jest wyrażeniem o wartości całkowitej, `stala1, stala2, ...`, są wyrażeniami stałymi o wartości całkowitej, a `lista1, lista2`, ..., są listami instrukcji (być może pustymi).

Instrukcja wyboru `switch`

- Wyrażeniem stałym całkowitym **`wyr_calk`** może być tu liczba podana w postaci literału, nazwa całkowitej zmiennej ustalonej lub wyrażenie całkowite składające się z tego typu podwyrażeń.
- Liczba fraz **`case`** może być dowolna.
- Stałe `stala1, stala2, ...`, występujące w każdej z fraz **`case`** muszą być różne.
- Listy instrukcji `lista1, lista2`, ... mogą też być puste.
- Fraza **`default`** jest opcjonalna: jeśli występuje, to może wystąpić tylko raz, choć niekoniecznie na końcu.

Instrukcja wyboru `switch` - działanie

- Najpierw obliczane jest **`wyr_calk`**.
- Następnie, jeśli obliczona wartość jest równa wartości którejś ze stałych `stala1, stala2, ...`, to wykonywane są instrukcje ze wszystkich list instrukcji, poczynając od listy we frazie **`case`** odpowiadającej tej stałej. (*A więc wykonywane są nie tylko instrukcje z listy w znalezionej frazie **`case`**, ale również ze wszystkich dalszych list!*)
- Jeśli żadna ze stałych `stala1, stala2, ...,` nie jest równa wartości **`wyr_calk`**, a fraza **`default`** istnieje, to wykonywane są wszystkie instrukcje poczynając od tych we frazie `default`.
- Jeśli natomiast żadna ze stałych `stala1, stala2, ...,` nie jest równa **`wyr_calk`**, a fraza **`default`** nie istnieje, to wykonanie całej instrukcji wyboru uznaje się za zakończone.

Instrukcja wyboru switch z break

- Dowolną z instrukcji może być instrukcja zaniechania(przerwania) **break**. Jeśli sterowanie przejdzie przez tę instrukcję, to wykonanie całej instrukcji wyboru kończy się, **break** przerywa instrukcję **switch**.
- Jeśli więc chcemy aby dla wyr_calk równego pewnej stałej wykonana była tylko lista instrukcji znajdujących się przy tej stałej to piszemy:

```
switch(wyr_calk )
{
  case wyrażenie_stałe1: instrukcjeA ;
                         break;
  case wyrażenie_stałe2: instrukcjeB ;
                         break;
  //...
  default:  instrukcjeN ;
}
```

Instrukcja wyboru switch - przykład

```
switch(nr)
{
    case 3: cout << "*";
    case 2: cout << "-";
    case 1: cout << "!";
}
```

dla nr = 3: *-!

dla nr = 2: -!

dla nr = 1: !

dla innego nr: nic się nie wydrukuje

Instrukcja wyboru switch - przykład

```
switch(nr)
{
    case 3: cout << "*"; break;
    case 2: cout << "-"; break;
    case 1: cout << "!"; break;
}
```

dla nr = 3: *

dla nr = 2: -

dla nr = 1: !

dla innego nr: nic się nie wydrukuje

Instrukcja wyboru switch - przykład

Napisz instrukcje która na podstawie zmiennej całkowitej ocena wyświetla jedną z informacji: *brak promocji do następnej klasy, promocja do następnej klasy, promocja z ocena celująca*

```
switch (ocena)
{
    case 1: cout << " brak promocji "; break;
    case 2:
    case 3:
    case 4:
    case 5: cout << " promocja do nastepnej klasy ";
            break;
    case 6: cout << " promocja z ocena celujaca";
            break;
    default: cout << "Bledny numer oceny";
}
```

Instrukcje iteracyjne - pętla `while`

Iteracyjna instrukcja sterująca **`while`** pozwala na realizację tak zwanej **pętli programowej** (ang. *iterative statement, loop*), polegającej na tym, że pewna instrukcja lub blok instrukcji wykonywane są „w kółko”, dopóki spełniony jest pewien warunek logiczny (wyrażenie ma wartość `true` lub w przypadku wyrażeń całkowitoliczbowych wartość jest `!=0`).

Instrukcja sterująca **`while`** ma formę:

```
while( wyr )
  instrukcja1;
```

Oczywiście **`instrukcja1`**, może być instrukcją złożoną (grupującą, blokiem instrukcji), czyli kilka instrukcji ujętych w klamry { }.

Instrukcje iteracyjne - pętla `while`

Wykonanie instrukcji **`while`:**

Najpierw obliczana jest wartość wyrażenia **`wyr`**

- Jeśli **`wyr`** ma wartość ***`false`***, wówczas **`instrukcja1`** nie jest wcale wykonywana.
- Jeśli jednak **`wyr`** ma wartość ***`true`***, wówczas wykonywana jest **`instrukcja1`**, po czym ponownie sprawdzany jest warunek tj. obliczana wartość **`wyr`**.
  - Jeśli tym razem **`wyr`** nadal jest *`true`*, wówczas ponownie wykonywana jest **`instrukcja1`** – i tak dalej, wielokrotnie, dopóki (*while*) **`wyr`** ma wartość ***`true`***.
  - Jeśli w końcu, za którymś obiegiem pętli, warunek stanie się ***`false`***, wówczas dopiero praca pętli zostanie przerwana.

Instrukcje iteracyjne - pętla `while`

Podsumowując:

```
while(wyr) instrukcja1;
```

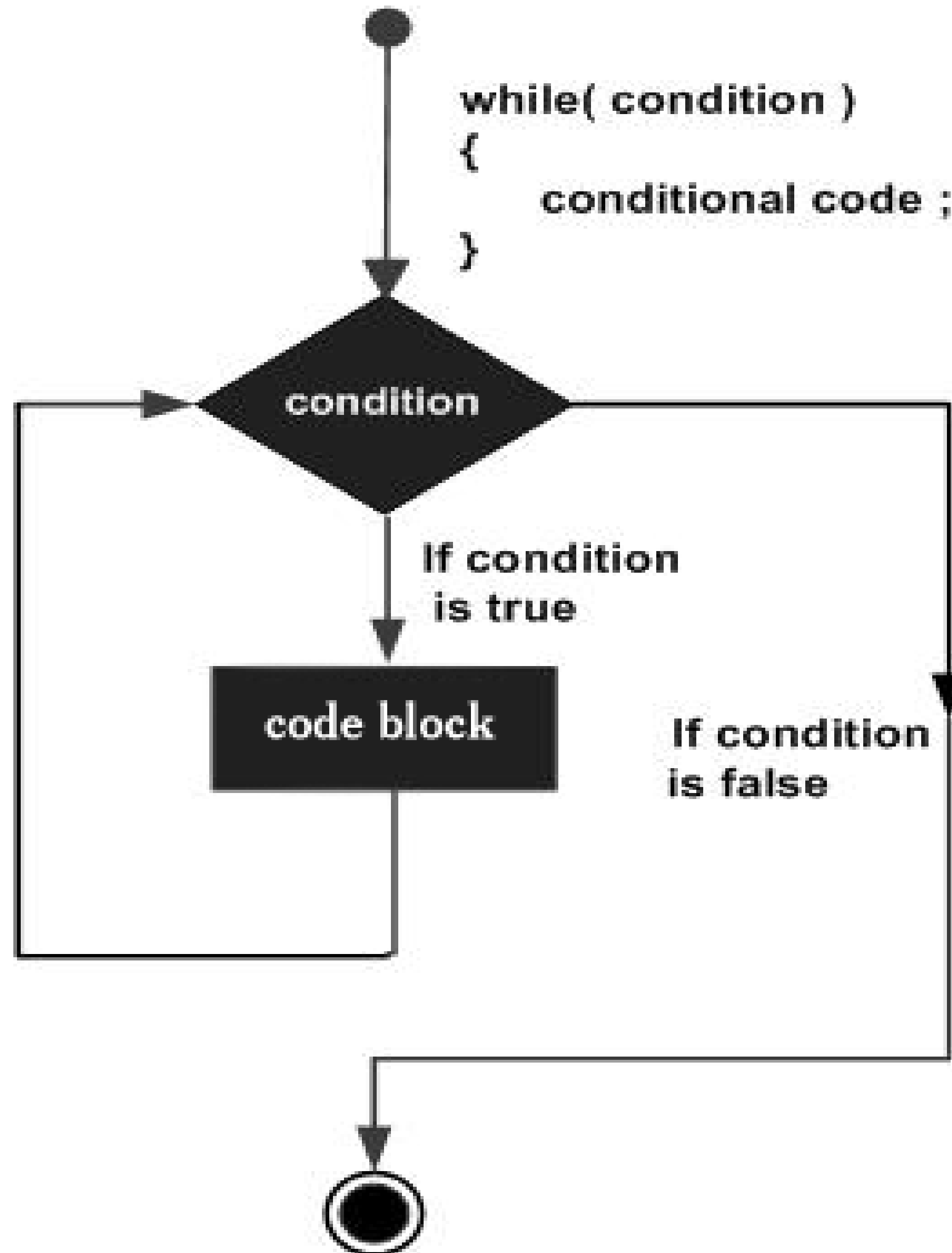
dopóki prawdziwe jest wyr wykonuj instrukcja1

instrukcja jest powtarzana, dopóki **`wyr`** jest prawdziwe (`true`). Obliczenie wartości wyrażenia następuje przed każdym kolejnym wykonaniem instrukcji.

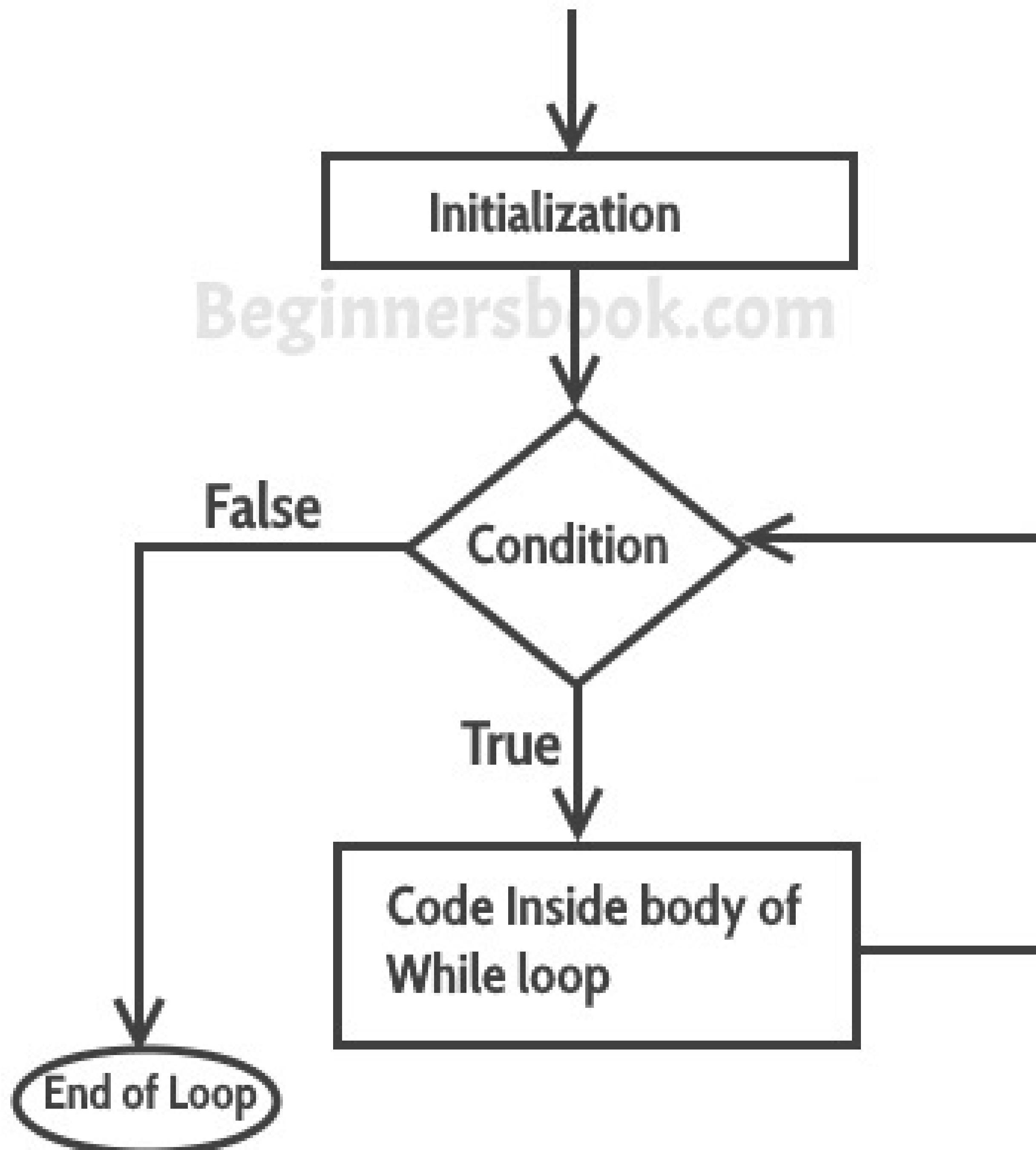
Zwróćmy uwagę, że pierwsze obliczenie wartości **`wyr`** odbywa *się* przed wykonaniem instrukcji **`instrukcja1`**, czyli możliwa jest sytuacja, gdy **`instrukcja1`** nie zostanie wykonana ani razu.

Ponadto przed pętlą **`while`** należy dokonać przygotowania: inicjalizacji zmiennych występujących w wyrażeniu **`wyr`**.

Instrukcje iteracyjne - pętla `while`



Instrukcje iteracyjne - pętla `while`



Pętla while – przykład 1 - gwiazdki

Program pobiera od użytkownika liczbę i drukuje tyle gwiazdek

```
int main()
{
  int ile;
  cout << "Podaj ile gwiazdek chcesz wyswietlic : ";
  cin >> ile;
  cout << "Narysujmy "<< ile <<" gwiazdek: ";
  // petla while rysujaca gwiazdki
  while(ile != 0) //lub: while(ile)
  {
     cout << '*'; //wyswietlamy gwiazdke
     ile--;  //ile = ile - 1; - zmniejszamy licznik
  }
  cout << "\nTeraz zmienna ile ma wartosc " << ile;
}
```

```
Podaj ile gwiazdek chcesz wyswietlic : 5
Narysujmy 5 gwiazdek: *****
Teraz zmienna ile ma wartosc 0
```

Pętla while – przykład 1

Zauważmy, że jeśli ktoś poda liczbę 0 to pętla nie wykona się, a jeśli liczbę ujemną to pętla będzie nieskończona:

```
while(ile!=0)//wyrażenie będzie zawsze true
{
   cout << '*';//wyswietlamy gwiazdke
   ile = ile - 1;//jeszcze mniejsza ujemna
}
```

Program z powyższym warunkiem powinien zatem przyjmować jedynie liczbę dodatnią.

Możemy zmienić warunek na `ile > 0` (wówczas dla liczy ujemnej pętla się nie wykona ani razu), albo wymusić podanie liczby dodatniej.

Poprawmy nasz program: jeśli użytkownik poda liczbę ujemną lub 0, to powinien zostać poinformowany, że wymagana jest liczba dodatnia i poproszony o kolejną liczbę. Próbę wczytywania liczby powtarzamy dopóty, dopóki użytkownik nie poda liczby poprawnej (dodatniej).

Taki sposób zapewnienia poprawności wczytywanych danych nazywać będziemy **pętlą zaporową**.

Pętla while – przykład 1 cd – pętla zaporowa

```
int main()
{
    int ile;
    cout << "Podaj ile gwiazdek chcesz narysowac : ";
    cin >> ile;
    //petla zaporowa:
    //dopoki ile<=0 pobieramy ponownie
    while (ile <= 0)  //while(!(ile>0))
    {
        cout << "Podales liczbę ujemna lub 0!"
             << endl << "Podaj liczbe dodatnia ";
        cin >> ile;
    }
    cout << "Liczba dodatnia wynosi:" << ile << endl;
    //teraz mozemy już wyswietlac ile gwiazdek
    cout << "Narysujmy "<< ile <<" gwiazdek: ";
    //...
```

Pętla do...while...

Pętla `do...while...` czyli: `rób... dopóki...`

Instrukcja ta pozwala na realizację innego rodzaju pętli programowej:

```
do
  instrukcja
while( wyr );
```

czyli :

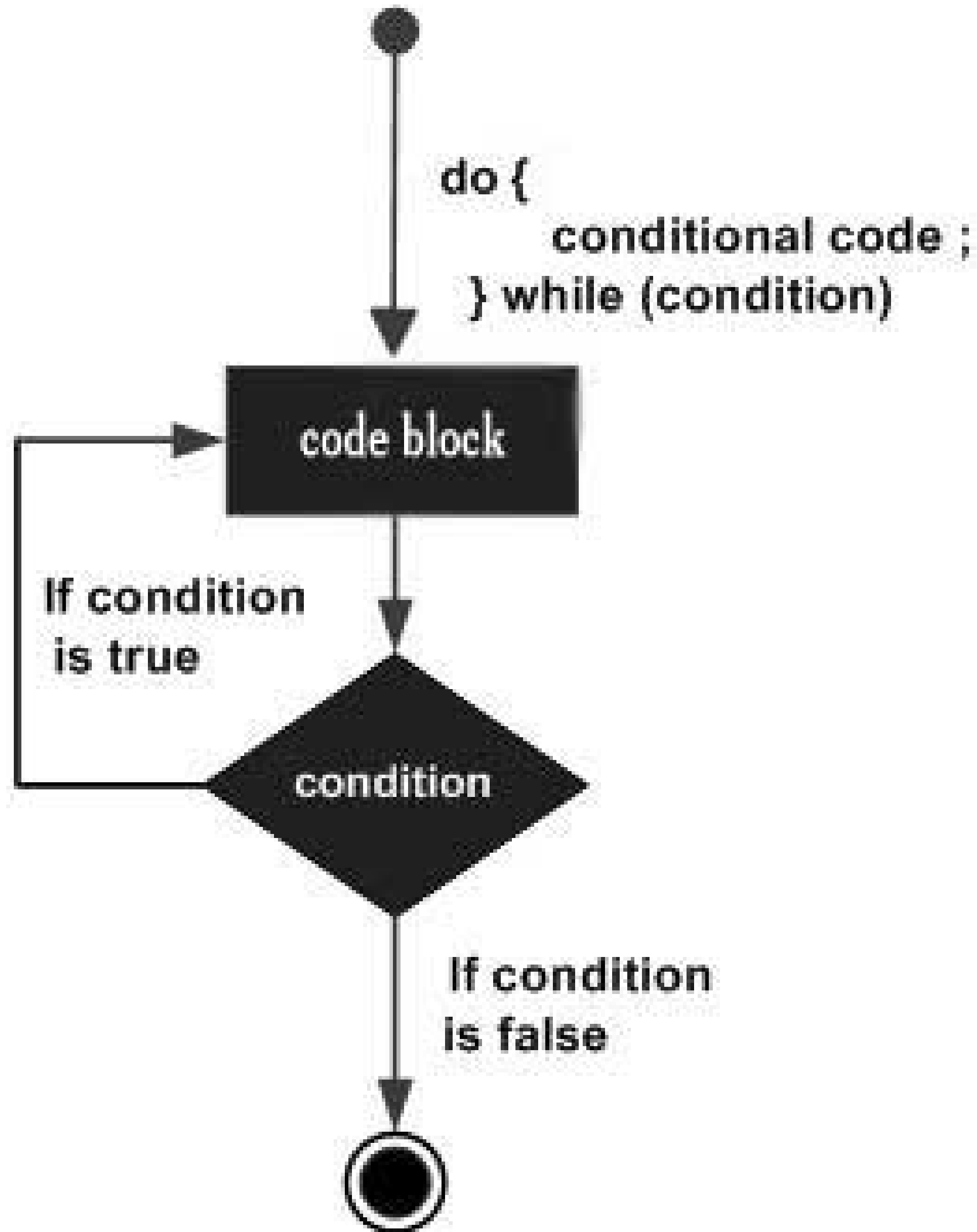
```
wykonuj instrukcja1 dopóki ( wyr );
```

Działanie jej jest takie:

Najpierw wykonywana jest `instrukcja1`.

Następnie sprawdzana jest wartość `wyr`. Jeśli `wyr` ma wartość `true` to wykonanie `instrukcja1` zostanie powtórzone, po czym znowu sprawdzany jest warunek `wyr`... i tak w kółko, dopóki warunek będzie spełniony(prawdziwy).

Pętla do...while...



Pętla do...while...

Jak widać, działanie tej pętli `do...while` przypomina działanie pętli `while...`

Różnica polega tylko na tym, że wyrażenie logiczne `wyr` obliczane jest nie przed, ale po wykonaniu `instrukcji1`.

Wynika stąd, że `instrukcja1` zostanie wykonana co najmniej raz. Czyli nawet wtedy, gdy `wyr` nie będzie nigdy prawdziwe (warunek nie będzie nigdy spełniony).

Przykład. Program wczytuje i wyświetla napisanie przez użytkownika litery, dopóki nie podamy litery *K* (wielkiej). Wtedy to wykonywanie pętli zakończy się. Zwróćmy uwagę, że pętla wczytująca znaki zostanie wykonana przynajmniej raz, zatem możemy wykorzystać pętle `do..while`.

Pętla do...while...

```
#include <iostream>
using namespace std;
int main()
{
    char litera;
    do {
        cout << "Napisz jakas litere: ";
        cin >> litera;
        cout << "Napisales: " << litera << " \n";
    }while(litera != 'K');
    cout << "Skoro napisales K to konczymy !";
    return 0;
}
```

Pętla do...while

Przykładowe wykonanie:

```
Napisz jakas litere: a
Napisales: a
Napisz jakas litere: A
Napisales: A
Napisz jakas litere: K
Napisales: K
Skoro napisales K to konczymy !
```

Pętla while i do...while -przykład – petla zaporowa

Napisz program wczytujący od użytkownika z klawiatury przy pomocy ”pętli zaporowej” liczbę całkowitą dodatnią.

```
//z użyciem pętli while
int main()
{
    int n;
    cout << "Podaj liczbe dodatnia";
    cin >> n; //pierwsze pobranie przed pętlą
    while (n <= 0) //dopóki liczba !(n>0) pobieramy ponownie
    {
        cout <<"Podales liczbę <=0 !"
             <<"\nPodaj liczbe dodatnia ";
        cin >> n;
    }
    cout << "Liczba dodatnia wynosi:" << n << endl;
    return 0;
}
```

Pętla while i do...while -przykład – petla zaporowa

```
//z użyciem pętli do...while
#include <iostream>
using namespace std;
int main()
{
    int n;
    do
    {
        cout << "Podaj liczbe dodatnia";
        cin >> n;
    }while (n <= 0);//!(n>0)
    cout << "Liczba dodatnia wynosi:" << n
         << endl;
    return 0;
}
```

Operator przypisania = *(grupa 16, wiązanie prawe)*

Operator przypisania **=**

```
double m;
m = 34.88;
```

powoduje że do obiektu stojącego po jego lewej stronie przypisana (podstawiona) zostaje wartość wyrażenia stojącego po prawej. Zatem w zmiennej `m` znajdzie się liczba `34.88`

Jest to operator dwuargumentowy o wiązaniu prawym.

Ponadto, każde przypisanie samo w sobie jest także wyrażeniem mającym taką wartość, jaka jest przypisywana:

```
int a, x = 4;
cout << "Wart. wyrazenia przypisania: " << (a = x);
```

W rezultacie na ekranie pojawi się napis:

```
Wart. wyrazenia przypisania: 4
```

Operator przypisania = *(grupa 16, wiązanie prawe)*

Dzięki temu, że wartością całego wyrażenia z przypisaniem jest wartość lewej strony po jego wykonaniu, przypisania można stosować kaskadowo:

```
int k = 7, j, m;
int n = m = j = k;
```

ponieważ **wiązanie operatora przypisania jest od prawej**, wartością wyrażenia '`m = j = k`', równoważnego '`m = (j = k)`', jest wartość ***m*** po przypisaniu (czyli w naszym przypadku 7). Ta wartość zostanie użyta do zainicjowania definiowanej zmiennej ***n***. Efektem ubocznym będzie nadanie wartości również zmiennym ***m*** i ***j***. Instrukcja byłaby błędna, gdyby któraś ze zmiennych ***m***, ***j***, ***k*** nie była utworzona wcześniej.

Operator przypisania = *(grupa 16, wiązanie prawe)*

Może się zdarzyć, że po obu stronach operatora przypisania stać będą argumenty różnego typu. Nastąpi wówczas próba niejawnej zamiany typu wartości przypisywanej na typ zgadzający się z typem tego, co stoi po lewej stronie:

```
int a;

a = 3.14;

cout << a; //3
```

Nastąpi tu zamiana (mówimy konwersja) typu zmiennoprzecinkowego na typ `int`. Po prostu zostanie wzięta pod uwagę tylko część całkowita liczby 3.14 – czyli 3 – i to zobaczymy na ekranie.

Konwersja taka nastąpi niejawnie.

Złożone operatory przypisania *(grupa 16, wiązanie prawe)*

Poznaliśmy już wcześniej operator przypisania =. W zasadzie może on nam wystarczyć, jednak dla wygody mamy jeszcze do dyspozycji następujące operatory:

```
+=   -=   *=   /=   %=
```

Są to **złożone operatory przypisania**.

Pozwalają na prostszy zapis niektórych przypisań: tych, w których ten sam obiekt występuje po lewej i prawej stronie przypisania.

Zamiast instrukcji

```
    a = a @ b;
```

gdzie symbol '@ ' oznacza któryś z operatorów :`+, -, *, /, %,` można użyć instrukcji

```
    a @= b;
```

Złożone operatory przypisania *(grupa 16, wiązanie prawe)*

Zatem np.

```
n += 2    oznacza    n = n + 2

n -= 2    oznacza    n = n - 2

n *= 2    oznacza    n = n * 2

n /= 2    oznacza    n = n / 2

n %= 2    oznacza    n = n % 2
```

Drobna różnica, najczęściej bez znaczenia, pomiędzy tymi instrukcjami polega na tym, że w przypisaniu złożonym wartość *a* jest obliczana jednokrotnie, a w zwykłym dwukrotnie.

Zwykle forma ' **`a @= b`**', jest efektywniejsza.

Pętle – przykład – suma cyfr liczby

Przykład. Napisz program, który dla wczytanej z klawiatury liczby całkowitej oblicza sumę cyfr tej liczby

Przykład rozkład liczby na kolejne cyfry l.naturalnej dla `n=125`

```
125 % 10 = 5 -ostatnia cyfra
125 / 10 = 12-pozbywamy się ostatniej cyfry
12 % 10 = 2  -kolejna „ostatnia” cyfra
12 / 10 = 1  -pozbywamy się ostatniej cyfry
1 % 10 = 1   -kolejna „ostatnia” cyfra
1 / 10 = 0   -koniec
```

Algorytm obliczania sumy cyfr liczby n>=0:

Dopóki n>0 (są jeszcze cyfry)

- obliczamy ostatnia cyfra (%10) i dodajemy ją do sumy,
- pozbywamy się ostatniej cyfry (/10)

Pętle – przykład – suma cyfr liczby

Zauważmy, że dla liczb ujemnych operator % zwraca wartość ujemną *np. dla* `n=-125` mamy `-125 % 10 = -5`

Cyfry liczby powinny być nieujemne, zatem dla liczb ujemnych będziemy wykonywać obliczenia na ich wartości bezwzględnej:

```
int n, rob;
cout << "Podaj liczbe calkowita ";
cin  >> n;
//funkcja-wartosc bezwzględna abs():
rob = abs(n);  // należy dodać: #include <cmath>
//lub
rob = (n<0)? -n : n; //operator warunkwy
//lub instrukcja selekcji
if(n<0) rob = -n;
else rob = n;
```

Pętle – przykład – suma cyfr liczby

```
int main()
{
    int n;
    cout << "Podaj liczbe calkowita ";
    cin  >> n;
    //jeśli n<0 to n%10<0, a cyfry liczby>0, to rob = abs(n)
    int  rob = (n<0)? -n : n; //w rob  wartość bezwzględna z n
    int s = 0;  //w zmiennej s będziemy przechowywać sumę cyfr
    while(rob > 0)  //dopóki są jeszcze cyfry
    {
       s += rob % 10;//do sumy + ostatnią cyfrę liczby rob
       rob /= 10;  //liczbę rob dzielimy przez 10,
        //skracamy w ten sposób ostatnia cyfrę
    }
    cout << "Suma cyfr liczby " << n
         << " wynosi " << s << endl;
    return 0;
}
```

Pętle – przykład – suma cyfr liczby

Output dla `n=-125` :

```
Podaj liczbe calkowita -125
Suma cyfr liczby -125 wynosi 8
```

Zauważmy, że dla `n=0`, pętla nie wykona się ani razu. Ale suma cyfr liczby `0` jest równa `0`, a tyle wynosi wartość zmiennej `suma` przed pętlą.

```
Podaj liczbe calkowita 0
Suma cyfr liczby 0 wynosi 0
```

Pętle – przykład – suma cyfr liczby

Przykład. Napisz program, który dla wczytanej z klawiatury liczby całkowitej oblicza ile cyfr ma wczytana liczba.

Zauważmy, że `n=0`, ma jedną cyfrę, zatem jeśli chcielibyśmy skorzystać z pętli z powyższego przykładu, należy najpierw sprawdzić czy wczytana liczba nie jest zerem (`if` przed pętlą `while`), lub użyć pętli `do-while`, która wykona się przynajmniej raz.

Pętle – przykład – liczba cyfr liczby- pętla while

```
int main()
{
    int n;
    cout << "Podaj liczbe calkowita n=";
    cin  >> n;
    int rob = abs(n);//można rob=n; bo tylko liczymy cyfry
    int ilcyfr = 0;//licznik cyfr liczby
    if ( rob == 0 )  //liczba n=0
        ilcyfr = 1;  //zero ma 1 cyfra
    else  //liczba !=0
        while (rob != 0)  //dopoki są cyfry
        {
            ilcyfr++;//rob!=0 ma cyfrę(y) -> zwiększamy licznik
            rob /= 10; //pozbywamy się już policzonej cyfry
        }
    cout << "Liczba " << n << " ma " << ilcyfr << " cyfr(y).";
    return 0;
}
```

Pętle – przykład – liczba cyfr liczby - pętla do-while

```
#include <iostream>
#include <cmath>  //bo używamy funkcji abs
using namespace std;
int main()
{
    int n;
    cout << "Podaj liczbe calkowita ";
    cin  >> n;
    int rob = abs(n);//można rob=n; bo tylko liczymy cyfry
    int il = 0; //zmienna - licznik ilości cyfr
    do
    {
        il++; //il = il+1;  il += 1; zwiększamy licznik
        //bo mamy pierwszą lub kolejna cyfrę liczby
        rob /= 10; //n = n/10;
    } while (rob != 0);
    cout << "Liczba cyfr liczby " << n << " to " << il ;
    return 0;
}
```