6. Instrukcje sterujące przebiegiem wykonania programu.

W tej części kursu poznamy podstawowe instrukcje języka C++ służące do sterownia przebiegiem wykonania programu. W języku C++ istnieją w zasadzie jedynie dwa typy instrukcji sterujących, są to *instrukcje warunkowe* oraz *pętle*. W ramach niniejszego kursu poznamy:

- dwa typy instrukcji warunkowych: **`if`** oraz **`if else`**,
- trzy typy pętli: pętla **`while`**, pętla **`do while`** oraz pętla **`for`**.

Każdy program napisany w języku C++, podobnie zresztą jak i w innych językach programowania, realizowany jest zgodnie z naturalną kolejnością instrukcji w nim zawartych. Występują jednak takie sytuacje w tworzonych algorytmach, które zapisywane są z użyciem języka programowania, w których ta naturalna kolejność musi być zaburzona, aby zapewnić prawidłowość działania implementowanego algorytmu. Sytuacje te najczęściej objawiają się wtedy gdy pewien ciąg instrukcji powinien być wykonany gdy spełniony jest określony warunek podczas działania programu. Równie często zdarza się sytuacja w której pewien ciąg instrukcji programu powinien być wielokrotnie powtórzony. W tych sytuacjach używamy odpowiednio: instrukcji warunkowych i pętli. Omówimy niżej dokładnie obydwa te typy instrukcji. Warto jeszcze dodać, że z teorii algorytmów wiemy, że te dwa typy instrukcji sterujących są w *zupełności wystarczające* do zapisu *każdego*, dowolnie skomplikowanego, algorytmu realizującego dowolne zadnie obliczeniowe (dowolną aplikację).

Przejdźmy najpierw do omówienia instrukcji warunkowych. Składnia tych instrukcji została podana niżej w Listingu 6.

```
1  //składnia instrukcji warunkowej if
2  if (wyrażenie)
3  {
4   instrukcja1; instrukcja2; ... instrukcjan;
5  }
6
7  //składnia instrukcji warunkowej if else
8  if (wyrażenie)
9  {
10  instrukcja1; instrukcja2; ... instrukcjan;
11 }
12 else
13 {
14  instrukcja1; instrukcja2; ... instrukcjam;
15 }
```

Listing 6. *Składnia instrukcji warunkowych języka C++.*

Instrukcja warunkowa **if** (pol. „jeśli") działa w ten sposób, że jeśli wartością wyrażenia liczbowego **wyrażenie** obecnego w nawiasie zwykłym jest jakakolwiek wartość różna od zera wówczas instrukcje w nawiasach klamrowych zawarte w linii 4 zostaną wykonane. W przeciwnym razie, tzn. gdy wartością wyrażenia jest *dokładnie* zero, instrukcje te zostaną pominięte i program będzie kontynuowany od pierwszej instrukcji po zamykającym nawiasie klamrowym obecnym w linii 5. Element składniowy **wyrażenie** może być dowolnym, poprawnym składniowo wyrażeniem liczbowym języka C++. Może to być np. liczba (stała) całkowita, zmienna rzeczywista czy np. (najczęściej) warunek logiczny. Zauważmy że warunek logiczny np. „x < 5" jest szczególnym przypadkiem wyrażenia liczbowego, ponieważ jego wartością jest w danym momencie wykonania programu wartość liczbowa 0 lub 1 w zależności od tego czy w momencie tym warunek „x < 5" jest spełniony (wartość 1 – tzn. prawda) czy też spełniony nie jest (wartość 0 – tzn. fałsz).

Instrukcja warunkowa **if else** (pol. „jeśli ... w przeciwnym razie") działa podobnie, z tą różnicą, że w przypadku gdy wartością wyrażenia jest zero wówczas instrukcje znajdujące się pomiędzy drugą parą nawiasów klamrowych (linie 13 – 15) zostaną wykonane, zaś instrukcje zawarte między pierwszą parą nawiasów klamrowych (linie 9 – 11) zostaną pominięte.

Aby dokładniej zrozumieć różnicę pomiędzy dwoma typami przedstawionych instrukcji warunkowych posłużmy się poniższym przykładem.

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main()
6  {
7   int liczbaPunktow, ocena;
8   cout << "Podaj liczbe punktow ";
9   cin >> liczbaPunktow;
10  if (liczbaPunktow >= 90) { ocena = 5; }
11  if (liczbaPunktow >= 80 and liczbaPunktow < 90) { ocena = 4; }
12  if (liczbaPunktow >= 70 and liczbaPunktow < 80) { ocena = 3; }
13  if (liczbaPunktow <  70) { ocena = 2; }
14  cout << "Uzyskana ocena wynosi " << ocena << endl;
15  return 0;
16 }
```

Listing 7. *Przykład użycia instrukcji warunkowych języka C++.*

Program prosi użytkownika o wprowadzenie liczby punktów jaką uczeń uzyskał z klasówki, a następnie przelicza tę liczbę na ocenę uzyskaną przez ucznia i wyświetla obliczoną ocenę w oknie konsoli. Stosujemy tu pierwszy wariant instrukcji warunkowej, tzn. instrukcję **if**. Zauważmy jednak że program, choć działa prawidłowo, nie jest napisany efektywnie. Wynika to z faktu, że np. jeśli uczeń uzyskałby 95 punktów wówczas warunek obecny w linii 10 byłby spełniony i zmienna **ocena** ustawiana zostałaby na wartość 5. Mimo, że w takim przypadku nie zachodzi już konieczność dalszego sprawdzania kolejnych kryteriów ocen, to sprawdzanie

takie jest nadal wykonywane w następnych liniach programu z użyciem instrukcji warunkowych **if**. Każde dodatkowe (niepotrzebne z logicznego punktu widzenia) sprawdzenie kolejnego kryterium oceny z użyciem instrukcji **if** zajmuje czas procesora, co powoduje, że program będzie działał wolniej niż gdyby został napisany w bardziej efektywny sposób. Spróbujmy zatem zapisać ten sam algorytm w bardziej efektywny sposób z użyciem drugiego wariantu instrukcji warunkowej.

```
#include <iostream>

using namespace std;

int main()
{
 int liczbaPunktow, ocena;
 cout << "Podaj liczbe punktow ";
 cin >> liczbaPunktow;
 if (liczbaPunktow >= 90)
 {
  ocena = 5;
 }
 else
 {
  if (liczbaPunktow >= 80)
  {
   ocena = 4;
  }
  else
  {
   if (liczbaPunktow >= 70)
   {
    ocena = 3;
   }
   else
   {
    ocena = 2;
   }
  }
 }
 cout << "Uzyskana ocena wynosi " << ocena << endl;
 return 0;
}
```

Listing 8. *Przykład użycia instrukcji warunkowych języka C++.*

Teraz sytuacja zmieniła się – program będzie działał w sposób identyczny jak ten z Listingu 7, jednak bardziej efektywnie, o czym zaraz się przekonamy. Znów rozważmy sytuację w której uczeń uzyskał ocenę 5. Wtedy warunek w linii 10 jest spełniony i wykonywana jest instrukcja w linii 12 zapisująca do zmiennej `ocena` wartość 5. Jednak teraz, po wykonaniu wspomnianego zapisu, blok instrukcji związany z członem `else` naszej instrukcji warunkowej, tzn. blok rozpoczynający się nawiasem klamrowym w linii 15 i kończący się nawiasem klamrowym w linii 31, zostaje pominięty i program kontynuuje działanie od instrukcji zawartej w linii 32 wyświetlającej komunikat o ocenie uzyskanej przez ucznia omijając kolejne sprawdzenia. Ta sama zasada dotyczy wszystkich pozostałych instrukcji `if else` zagnieżdżonych w pierwszym członie warunkowym dotyczącym przypadku uzyskania oceny 5. Taka realizacja algorytmu obliczającego ocenę ucznia na podstawie liczby uzyskanych przez nią/niego punktów będzie działać niewątpliwie szybciej niż wersja z Listingu 7, ponieważ w momencie ustalenia wartości uzyskanej oceny dalsze sprawdzenia są pomijane.

Zajmijmy się teraz drugim z omawianych typów instrukcji sterujących przebiegiem wykonania programu jakimi są pętle. Pętle używane są w sytuacji gdy pewien ciąg instrukcji programu powinien być wykonany wielokrotnie. W zależności od konkretnej sytuacji mamy w języku C++ do wyboru trzy rodzaje pętli. Podajmy składnię pierwszego rodzaju pętli, tzn. pętli „while”.

```
1 while (wyrażenie)
2 {
3  instrukcja1; instrukcja2; ... instrukcjan;
4 }
```

Listing 9. *Składnia pętli „while” języka C++.*

Pętla „while” (pol. „podczas gdy”) działa w ten sposób, że jeśli wyrażenie jest różne od zera (w szczególności warunek logiczny jest prawdziwy) to wykonywane będą instrukcje w ciele pętli. Po wykonaniu ostatniej instrukcji w ciele pętli sterowanie zostanie przekazane na jej początek gdzie ponownie sprawdzona zostanie wartość wyrażenia. A więc instrukcje w ciele pętli będą wykonywane cyklicznie do momentu w którym wartość wyrażenia stanie się zerem (w szczególności do momentu w którym warunek logiczny będzie fałszywy). Zauważmy, że jeśli wyrażenie będzie miało wartość zero przed pierwszym sprawdzeniem tej wartości w pętli „while” to instrukcje w ciele pętli nie zostaną wykonane ani razu. Podajmy przykład programu wykorzystującego pętlę „while”.

```
1 #include <iostream>
2
3 using namespace std;
4
5 int main()
6 {
7  char imie[20];
8  int licznik = 0;
9  cout << "Podaj swoje imie ";
```

```
10  cin >> imie;
11  while (imie[licznik] != 0)
12  {
13   licznik = licznik + 1;
14  }
15  cout << "Twoje imie sklada sie z " << licznik << " znakow." << endl;
16  return 0;
17 }
```

Listing 10. *Przykład zastosowania pętli „while” w języku C++.*

Zadaniem programu jest wyznaczenie i wyświetlenie w oknie konsoli liczby znaków (liter) z których składa się imię użytkownika wprowadzone przez niego z klawiatury. W języku C++ łańcuchy znaków reprezentujące tekst zapisywane są w tablicach znakowych, przy czym obowiązuje umowa, że tekst kończy się znakiem o kodzie liczbowym 0 nawet jeśli definicja tablicy zawiera więcej znaków niż tekst w niej zawarty. Reszta znaków po znaku o kodzie liczbowym 0 w takiej tablicy uznawana jest za nieistotne i należy je pominąć we wszelkich operacjach tekstowych, takich jak np. wyświetlanie, przeszukiwanie tekstu itp. W pętli „while” z naszego przykładu porównujemy kolejne znaki w tablicy **`imie`** do momentu kiedy napotkanym znakiem będzie znak o kodzie 0. W każdym cyklu (tzw. *iteracji*) pętli zwiększamy wartość zmiennej **`licznik`** o 1, aby przejść w kolejnej iteracji pętli do sprawdzenia następnego znaku w analizowanym tekście. W ten sposób wartość zmiennej **`licznik`** będzie zwiększana o 1 w każdej iteracji pętli do momentu napotkania znaku o kodzie zero w tablicy **`imie`**, co spowoduje zakończenie działania pętli. Po zakończeniu jej działania zmienna **`licznik`** zawierać będzie właśnie liczbę znaków zawartych w tablicy **`imie`** (nie licząc znaku o kodzie 0 definiującym koniec tekstu zawartego w tablicy). Korzystając z ostatniego faktu wyświetlamy zawartość zmiennej **`licznik`** informując użytkownika o liczbie liter z jakich składa się wprowadzone przez niego z klawiatury imię. Na koniec zauważmy, że gdyby użytkownik wcisnął klawisz Enter w odpowiedzi na prośbę podania imienia, bez wprowadzenia żadnych innych znaków, wówczas wprowadzanie zakończyłoby się i pierwszym znakiem zawartym w tablicy **`imie`** byłby znak o kodzie zero (czyli z logicznego punktu widzenia tekst zawarty w tablicy byłby tekstem pustym, nie zawierającym żadnych znaków). W takim wypadku instrukcja zwiększenia wartości zmiennej **`licznik`** o 1, zawarta w pętli „while”, nie wykonałaby się ani razu. Wobec tego wartość zmiennej licznik pozostałaby równa zeru i otrzymalibyśmy komunikat „Twoje imie sklada sie z 0 znakow.”, co z logicznego punktu widzenia jest stwierdzeniem sensownym. Efekt ten jest możliwy do osiągnięcia właśnie z powodu faktu, iż warunek zawarty w pętli „while” sprawdzany jest na początku jej wykonania.

Przejdźmy do omówienia drugiego rodzaju pętli, tzn. pętli „do while” (pol. „wykonuj podczas gdy”). Składnia tej pętli podana jest w Listingu 11. Pętla ta działa analogicznie do pętli „while” z tą różnicą, że wartość wyrażenia określająca warunek zakończenia działania pętli sprawdzana jest na końcu, tzn. po wykonaniu instrukcji zawartych w ciele pętli. Z tego powodu pętla ta wykorzystywana jest w sytuacji gdy instrukcje w niej zawarte muszą być wykonane przynajmniej raz.

```
do
{
 instrukcja1; instrukcja2; ... instrukcjan;
}
while (wyrażenie);
```

Listing 11. *Składnia pętli „while" języka C++.*

Podajmy przykład programu wykorzystującego pętlę „do while".

```
#include <iostream>

using namespace std;

int main()
{
 char znak;
 do
 {
  cout << "Czy kontynuowac?";
  cin >> znak;
 }
 while (znak == 't');
 return 0;
}
```

Listing 12. *Przykład zastosowania pętli „do while" w języku C++.*

Program nieustannie pyta użytkownika czy ten chce kontynuować jego działanie. Jeśli użytkownik wciśnie klawisz 't' na klawiaturze (potwierdzając wybór wciśnięciem klawisza Enter) wówczas pytanie zostaje ponowione. W przeciwnym razie, tzn. gdy wciśnięty zostanie jakikolwiek inny klawisz różny od znaku 't', wówczas program zakończy się. Zauważmy, że w tym wypadku pytanie o kontynuację zostanie zadane przynajmniej raz, co wynika z faktu, że warunek dotyczący zakończenia działania pętli sprawdzany jest na końcu (po wykonaniu instrukcji w ciele pętli).

Ostatnim rodzajem pętli dostępnym w języku C++ jest pętla „for". Składnia tej pętli jest następująca:

```
for (instrukcja0; wyrażenie; instrukcjan+1)
{
 instrukcja1; instrukcja2; ... instrukcjan;
}
```

Listing 13. *Składnia pętli „for" języka C++.*

Pętla ta, w odróżnieniu od dwóch poprzednich, wykorzystywana jest najczęściej w sytuacjach gdy albo z góry wiadomo (tzn. już w trackie pisania kodu programu) ile razy instrukcje w ciele pętli powinny być wykonane, bądź wówczas gdy, choć trackie pisania kodu programu nie można przewidzieć ile razy dokładnie instrukcje w ciele pętli będą wykonane, liczba ta jest uzależniona jednoznacznie od innych danych, zmiennych zawartych w programie. Obydwie sytuacje zostaną przedstawione na poniższych dwóch przykładach.

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main()
6  {
7   int i;
8   for (i = 1; i <= 10; i++)
9   {
10   cout << i << endl;
11  }
12  return 0;
13 }
```

Listing 14. *Przykład zastosowania pętli „for” w języku C++.*

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main()
6  {
7   int N, i;
8   cout << "Ile liczb mam wyswietlic? ";
9   cin << N;
10  for (i = 1; i <= N; i++)
11  {
12   cout << i << endl;
13  }
14  return 0;
15 }
```

Listing 15. *Przykład zastosowania pętli „for” w języku C++.*

Obydwa zaprezentowane programy działają bardzo podobnie. Pierwszy z nich wyświetla w oknie konsoli liczby od 1 do 10-ciu włącznie, każdą w oddzielnej linii okna. Drugi różni się tylko tym, że górny zakres liczb do wyświetlenia podawany jest przez użytkownika i zapisywany w zmiennej **N**. Zmienna ta jest następnie użyta w warunku określającym moment zakończenia pętli. Przykłady te obrazują sytuacje omówione na wstępie. W pierwszym przy-

padku instrukcje zawarte w ciele pętli wykonają się dokładnie 10 razy i fakt ten jest znany programiście w momencie tworzenia kodu programu. W drugim przypadku nie jesteśmy w stanie przewidzieć bezpośrednio ile razy pętla „for" wykona się (zależy to od wartości zmiennej **N**, która wprowadzana jest przez użytkownika), ale wiemy że wykona się ona dokładnie *N* razy jeśli wartością zmiennej **N** będzie liczba *N*. Warto dodać, że pętla for z racji swej konstrukcji bardzo często wykorzystywana jest do iterowania (czyli „przechodzenia") po elementach tablic.