



Programowanie zaawansowane FM i NI

Wykład 2

Pętle, warunki, funkcje, rand

Krzysztof Piasecki

Semestr letni roku akad. 2024-25



- **Pętle while i do..while**

Ta zmienna będzie pełnić rolę indeksu, po którym iterujemy kroki pętli. Przypisujemy wartość startową.

Sprawdza prawdziwość warunku **na początku** każdego kroku (dokładniej: czy nie-fałsz)

Pojedynczy krok pętli

[\[Link\]](#)

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

```
1 #include <iostream>
2 using namespace std;
3
4 int main ()
5 {
6     int i = 5;
7
8     while ( i <= 20 ) {
9         cout << "i = " << i << endl;
10        i += 1;
11    }
12    return 0;
13 }
```

Otwarcie pętli do...while

Pojedynczy krok pętli

Sprawdza prawdziwość warunku **na końcu** każdego kroku (dokładniej: czy nie-fałsz)

- **Pętla for:** w nagłówku ma zawsze 3 pola, oddzielone średnikami.

[\[Link\]](#)

Pole inicjalizacji

Można tu ustawić zmienną. Można też zadeklarować, ale nie jest to konieczne.

Wykona się zawsze 1×, na początku pętli.

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

Pole iteracji

Zwykle zmieniamy tu wartość indeksu.

Wykona się, gdy kod dotrze do **końca** kroku.

Pole warunku

Sprawdza prawdziwość warunku (czy *nie-fałsz*)
na początku każdego kroku

- W polu iteracji użyliśmy `i++` . To **inkrementacja** (zwiększenie wartości zmiennej o 1). Można też tak: `i--` . To **dekrementacja** (zmniejszenie wartości zmiennej o 1).
- Można też zmienić wartość zmiennej o dowolną wartość:

`i += 5 ; i -= j ; i *= 5 - k ; i /= sqrt(4.) ; i %= m ;`

- Polecenia sterujące **break** i **continue**.

[\[Link\]](#)

```
1  #include <iostream>
2  using namespace std;
3
4  int main ()
5  {
6      int i = 0;
7      while ( true )
8      {
9          i++ ;
10         if ( i % 2 == 0 ) {
11             continue ;
12         }
13         if ( i == 9 )
14             break ;
15
16         cout << i << endl;
17     }
18     return 0;
19 }
```

continue;

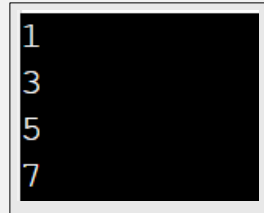
przerywa bieżący krok pętli
(ale nie całą pętlę).

O ile warunek logiczny jest spełniony,
to pętla kontynuuje bieg.

break;

przerywa całą pętlę.

Kod przechodzi poniżej końca.



```
1
3
5
7
```

Brak tu nawiasów { }. Tym razem to nie błąd.
C++ pozwala wyjątkowo opuścić { },
jeżeli w środku jest tylko 1 polecenie.

Uwaga: niebezpieczeństwo błędu w działaniu kodu.
Dbaj o czytelność, stosując wcięcie + Enter

- Instrukcja warunkowa
switch / case / default

switch poddaje testowi wartość zmiennej, ale tylko typu całkowitego

W **case** kodujemy postępowanie, jeśli zmienna ma konkretną wartość

Jeśli przypadki mają być rozłączne, to na końcu case musi być **break**.

W **default** kodujemy działanie, gdyby żaden case nie pasował.

Można też sprawdzać znak w zmiennej typu `char`.

Pranie mocne kolorowe.

```
1  #include <iostream>
2  using namespace std;
3
4  int main ()
5  {
6      int Program = 2 ;
7      switch (Program) {
8          case 1 : cout << "Pranie lekkie " ;
9                  break;
10         case 2 : cout << "Pranie mocne " ;
11                 break;
12         default: cout << "Brak programu" << endl;
13                 exit (0);
14     }
15     char Kolor = 'k' ;
16     switch (Kolor) {
17         case 'k' : cout << "kolorowe." << endl;
18                 break;
19         case 'b' : cout << "białe." << endl;
20                 break;
21         default : cout << "(wybierz kolor)" << endl;
22     }
23     return 0;
24 }
```

- **Funkcja:** to samodzielny blok kodu, który może przyjmować argumenty i zwraca coś (lub nic).
Uwaga: w C++ nie można umieścić funkcji w środku funkcji.
Funkcję zwykle **wywołuje się** (call) w danym miejscu, a wartość zwracana w to miejsce się „wkleja”.

[Link]

nazwa funkcji
typ, w jakim zostanie
zwrócony rezultat.

Nagłówek funkcji
(header)

```
1  #include <iostream>
2  using namespace std;
3
4  int reszta (int a, int b)
5  {
6      cout << " {obliczam} ";
7      return a % b ;
8  }
9
10 int main ()
11 {
12     int x = 7, y = 5 ;
13     cout << "x % y = "
14         << reszta (x, y) << endl;
15     return 0;
16 }
```

Tu deklarujemy zmienne,
tzw. „**argumenty wejścia**”

Ich zakres istnienia jest
ograniczony do funkcji.

Ciało funkcji (body)

return:
polecenie zwrotu

Miejsce **wywołania** funkcji.
Tu podajemy argumenty.
return z wywołanej funkcji
wklei rezultat w to miejsce.



x % y = {obliczam} 2

- Po funkcji main widać, że funkcja może nic nie przyjmować.
Wywołanie wyglądałoby tak: funkcja ();
- Ale może też nie zwracać.
Technicznie – zwraca wtedy **void** (ang. pustka).
void jest określeniem typu.

Ta funkcja zwraca void. Czyli – nic.

Można pominąć return
lub napisać:
return ;
(kończymy i nic nie zwracamy)

return nie musi być
na końcu kodu funkcji.

Tu funkcja powitanie się wykona.

Wyjątkowo w main wolno
nie napisać return .

```
Witaj. Podaj x i y: 7 0
{Dzielenie przez 0} nan
```

```
1  #include <iostream>
2  #include <cmath>
3  using namespace std;
4
5  void powitanie () {
6      cout << "Witaj. Podaj x i y: ";
7  }
8
9  float dzielenie (float a, float b) {
10     if (b == 0) {
11         cout << "{Dzielenie przez 0} " ;
12         return NAN;
13     }
14     return a / b ;
15 }
16
17 int main ()
18 {
19     float x, y ;
20     powitanie () ;
21     cin >> x >> y;
22     cout << dzielenie (x, y) << endl;
23 }
```


- Zakres widoczności zmiennej (scope)

- ① Każda zmienna zadeklarowana w obrębie {}, istnieje tylko w tych {}.
- ② Najbardziej zewnętrzne są **zmienne globalne** (przed {})
- ③ C++ dopuszcza, aby w {} wewnętrznym utworzyć zmienną o tej samej nazwie, co istniejąca zmienna w {} zewnętrznym. Ta zewnętrzna staje się „chwilowo niedostępna”. To tzw. **przesłanianie (shadowing)**.

Jest to **zła praktyka** (czytelność kodu...) ale trzeba rozumieć, jak zadziała C++.

```
[Main Start] 7
[Myfun A ] -2
[Myfun B ] 7
[Main { } ] 0
[Main for ] 15
[Main After] 7
```



```
1  #include <iostream>
2  using namespace std;
3
4  int A = -2;
5
6  void myfun (int B) {
7      cout << "[Myfun A ] " << A << endl;
8      cout << "[Myfun B ] " << B << endl;
9  }
10
11 int main ()
12 {
13     int A = 7;
14     cout << "[Main Start] " << A << endl;
15     myfun (A) ;
16     {
17         int A = 0;
18         cout << "[Main { } ] " << A << endl;
19         for (int A = 15; A <= 15; A++)
20             cout << "[Main for ] " << A << endl;
21     }
22     cout << "[Main After] " << A << endl;
23 }
```


- **Prototyp** funkcji a jej **implementacja**

1. Kompilator czyta kod od góry do dołu.
2. Wpierw deklaracja, później użycie.

Czy to znaczy, że funkcje muszą być zawsze umieszczane powyżej miejsca ich użycia?

Prototyp (nagłówek funkcji zakończony ;) to „zapowiedź” funkcji, wystarczająca aby funkcję umieścić gdzie indziej.

Implementacja (nagłówek i ciało) można umieścić gdziekolwiek poniżej prototypu.

Uwaga:

w prototypie, w nawiasie nie trzeba podawać nazw argumentów wejścia. Wymagane są typy.

```

1  #include <iostream>
2  #include <cmath>
3  using namespace std;
4
5  void powitanie () ;
6  float dzielenie (float, float) ;
7
8  int main () {
9      float x, y ;
10     powitanie () ;
11     cin >> x >> y;
12     cout << dzielenie (x, y) << endl;
13     return 0;
14 }
15
16 void powitanie () {
17     cout << "Witaj. Podaj x i y: ";
18 }
19
20 float dzielenie (float a, float b) {
21     if (b == 0) {
22         cout << "{Dzielenie przez 0} " ;
23         return NAN;
24     }
25     return a / b ;
26 }
```

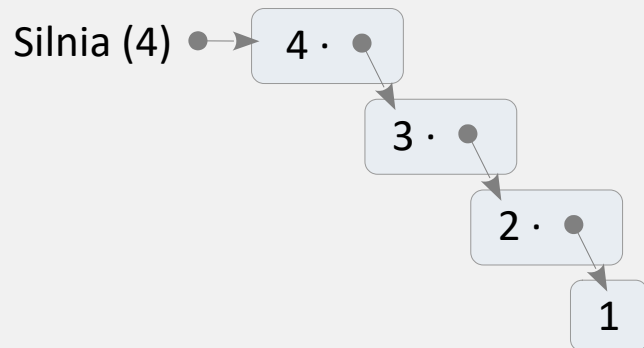
- **Rekurencyjność** funkcji: może ona wywoływać siebie samą.
Stan dotychczasowej funkcji „się zamraża”, dopóki komputer nie wróci z tej nowo-wywołanej.

[\[Link\]](#)

Do rekurencji nadaje się np. silnia:

$$N! = N \cdot (N-1)!$$

Miejsce wywołania rekurencyjnego



Funkcja rekurencyjna nie może być wywoływana w ∞ .
Musi mieć punkt końcowy.

```
Podaj N: 5
N! = 120
Podaj N: 17
N! = -288522240
Podaj N: -1
```

ciekawe... ☺

```
1  #include <iostream>
2  using namespace std;
3
4  int silnia (int N) {
5      if (N == 0 || N == 1)
6          return 1;
7      else
8          return N * silnia (N - 1) ;
9  }
10
11 int main () {
12     int N;
13     while (true) {
14         cout << "Podaj N: ";
15         cin >> N;
16         if (N >= 0)
17             cout << "N! = " << silnia ( N ) << endl;
18         else
19             return 0;
20     }
21 }
```

- Funkcje **przeciążone** (**overloaded**): można zakodować funkcje o tej samej nazwie, ale różniące się typami argumentów wejścia i/lub ich liczbą. Są to wówczas zupełnie różne funkcje.

[\[Link\]](#)

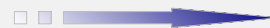
```
1  #include<iostream>
2  using namespace std;
3
4  double myfun (int    i) { return 1.; }
5  double myfun (char   c) { return 2.; }
6  double myfun (double d) { return 3.; }
7
8  int main() {
9      int    iinput = 7   ;
10     char   cinput = 72  ; // (char) 72 == 'H'
11     double dinput = 72.5 ;
12
13     cout << "myfun (int    ) : " << myfun (iinput) << endl;
14     cout << "myfun (char   ) : " << myfun (cinput) << endl;
15     cout << "myfun (double): " << myfun (dinput) << endl;
16 }
```

funkcje
przeciążone

Komputer zorientuje się
po typie
podanego argumentu,
którą funkcję
ma wywołać

Uwaga: nie mogą istnieć funkcje różniące się jedynie typem zwracanym. Kompilator nie wiedziałby, którą ma wywołać.

```
myfun (int    ) : 1
myfun (char   ) : 2
myfun (double): 3
```



- Podstawowa generacja **liczb pseudolosowych**

Podstawową funkcją jest `rand()` w `<cstdlib>`. Zwraca ona „losowe” inty $\in \{0, 1, \dots, \text{RAND_MAX}\}$. `RAND_MAX` na „typowym PC” – to ok. $2.15 \cdot 10^9$.

- Gdy kilkakrotnie włączymy program z generacją, zobaczymy, że sekwencja liczb się powtarza. Przyczyną jest, że `rand` startuje od tego samego **Ziarna**. Aby tego uniknąć, trzeba 1× zainicjować generator ziarnem „losowym”. Recepta:

```
#include <ctime>
```

```
...  
srand ( time(NULL) ) ;
```

Ziarnem będzie liczba sekund od 1.01.1970.

- Jak wygenerować losowy int z przedziału $[a, b)$?

→ np.: `int val = a + rand() % (b-a) ;`

- Jak wygenerować losowy float $\in [0, 1]$?

→ np.: `float val = rand() / float(RAND_MAX) ;`

- Jak wygenerować losowy float $\in [a, b]$?

→ np.: `float val = a + (b-a) * (rand() / float(RAND_MAX)) ;`

```
1 #include <iostream>
2 #include <cstdlib>
3 #include <ctime>
4 using namespace std ;
5
6 int main ()
7 {
8     srand ( time (NULL) );
9     cout << "Losowy int: " << rand() << endl;
10    cout << "RAND_MAX wynosi " << RAND_MAX ;
11
12    cout << "\nKilka int'ow z [2..8) : " ;
13    for ( int proba = 1 ; proba < 7 ; proba++ )
14        cout << 2 + rand() % (8 - 2) << ' ' ;
15
16    cout << "\nKilka float'ow z [0..1] : " ;
17    for ( int proba = 1 ; proba < 4 ; proba++ )
18        cout << rand() / float( RAND_MAX ) << ' ' ;
19
20    cout << "\nKilka float'ow z [3..7] : " ;
21    for ( int proba = 1 ; proba < 4 ; proba++ )
22        cout << 3. + 4. * ( rand() / float( RAND_MAX ) )
23        << ' ' ;
24 }
```

- **Referencja** – to nowa nazwa na istniejącą zmienną
- Składnia na prostym przykładzie: (uwaga: &)

```
int A = 5 ;
int& Aref = A ;
Aref = 7;      (sprawi, że zarazem A = 7 )
```

Nie da się zadeklarować 'pustej' referencji:

```
--int& Bref--;
```

Nie da się 'przepiąć' referencji na inną zmienną. To:

```
int C = 5 ;    Aref = C ;
```

będzie znaczyć tylko tyle, że do A przypiszemy 5.

- Główna użyteczność:
przy przekazywaniu zmiennej do podfunkcji.

Bez referencji - wywołana funkcja działa na **kopii** zmiennej podanej w wywołaniu. Zmiany w tej kopii nie zmieniają oryginału.

Działając na referencji - działamy na **orygynie**. Tylko tak zmiana wartości zmiennej w funkcji będzie się liczyć po powrocie do miejsca wywołania.

```
10
12
```

[\[Link\]](#)

```
1  #include<iostream>
2  using namespace std;
3
4  void Add_pass_via_Copy (int X) {
5      X    += 2;
6  }
7
8  void Add_pass_via_Ref (int& Xref) {
9      Xref += 2;
10 }
11
12 int main () {
13     int x = 10 ;
14     Add_pass_via_Copy ( x );
15     cout << x << endl;
16
17     x = 10 ;
18     Add_pass_via_Ref  ( x );
19     cout << x << endl;
20 }
```