Wstęp do informatyki- wykład 5

Instrukcja selekcji `if-else`
Operatory – arytmetyczne i logiczne
Wyrażenie warunkowe – operator selekcji
Instrukcja `switch`

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.

Instrukcja warunkowa `if-else`

Instrukcja warunkowa (selekcji) (ang. *conditional statement*) występuje w dwóch postaciach. Prostsza to

```
if ( wyr ) instr
```

gdzie **`wyr`** jest wyrażeniem o wartości logicznej, a **`instr`** jest instrukcją.

Przypomnijmy, że w C/C++ również liczbowe wartości całkowite i wskaźnikowe mogą być traktowane jako wartości logiczne:

Wartość **zero** rozumiana jest jako: **fałsz**.

Wartość **inna niż zero** rozumiana jest jako: **prawda**.

*Działanie instrukcji sterującej **`if`**:*

najpierw sprawdzany jest warunek tj. obliczana jest wartość wyrażenia **`wyr`**:

- jeśli wynosi ona **`true`**, wykonywana jest instrukcja **`instr`**
- jeśli **`false`**, **`instr`** nie jest wykonywana,

Instrukcja warunkowa `if-else`

Zauważmy, że składnia mówi o jednej instrukcji **`instr`**.

Jeśli zachodzi potrzeba wykonania (lub zaniechania wykonania) większej liczby instrukcji, to należy ująć je w nawiasy klamrowe, tak aby uczynić z nich jedną **instrukcję złożoną** (grupującą)- blok

```
{
    instrukcja1;
    instrukcja2;
    instrukcja3;
}
```

Instrukcja warunkowa `if-else`

Druga forma instrukcji warunkowej to

```
if ( wyr) instr1
else      instr2
```

gdzie, jak poprzednio, **`wyr`** jest wyrażeniem o wartości logicznej, a **`instr1 i instr2`** są instrukcjami — prostymi lub złożonymi.

Wykonanie instrukcji warunkowej w tej formie polega na

- obliczeniu wartości **`wyr`**, a następnie:
  - jeśli **`wyr`** ma wartość **`true`**, to wykonywana jest **`instr1`**,
  - w przeciwnym razie (*else*), czyli gdy **`wyr`** ma wartość **`false`** (warunek nie był spełniony), zostanie wykonana **`instr2`**

I znów składnia mówi o jednej instrukcji **`instr1`** i jednej instrukcji **`instr2`**. Jeśli zachodzi potrzeba użycia kilku instrukcji, to należy, jak poprzednio, zastosować instrukcję złożoną.

`if-else` – instrukcja zagnieżdżona

Każda z instrukcji `instr1` i `instr2` może z kolei też być instrukcją warunkową.

Instrukcję **`if`** można zatem zagnieżdżać (wybór wielowariantowy), przy czym **`else`** odpowiada wówczas najbliższy poprzedzający go **`if`**, po którym nie było jeszcze **`else`** i który jest zawarty w tym samym bloku na tym samym poziomie zagnieżdżenia co ten **`else`**.

```
if (wyrażenie1) isntrukcja1;
else if (wyrażenie2) instrukcja2;
     else if (wyrażenie3) instrukcja3;
          else if (wyrażenie4) instrukcja4;
```

if-else – instrukcja zagnieżdżona

```
if(wyrA)
if(wyrB) instrukcja1;
else
    instrukcja2;
```

Do którego **if** należy to **else**? Zasada jest taka, że:

Jeśli nawiasy klamrowe nie stanowią inaczej, to **else** odnosi się zawsze do najbliższego poprzedzającego go **if**.

Poprawmy formatowanie, dodając tabulatory:

```
if(wyrA)
    if(wyrB) instrukcja1;
    else
        instrukcja2;
```

Przypomnijmy, że tabulatory o niczym nie decydują. Są to białe znaki, które kompilator ignoruje. Dla programisty jednak są bardzo pomocne, bo przy ich użyciu program staje się czytelniejszy.
O sposobie zagnieżdżania decydują nawiasy klamrowe lub powyższa zasada.

if-else – instrukcja zagnieżdżona

```
// Program sprawdza czy podana przez użytkownika
//liczba jest dodatnia, ujemna czy równa 0
int main()
{
  int n;
  cout << "Podaj liczbe calkowita: ";
  cin >> n;
  if ( n > 0)
     cout << "Podales liczba dodatnia: " << n <<endl;
  else //n<=0
     if (n == 0)
        cout << "Podales 0." << endl;
     else //n<0
        cout <<"Podales liczba ujemna: "<< n <<endl;

    return 0;
}
```

if-else –wybór wielowariantowy

```
if( wyr1 ) instr1 ;
else if ( wyr2 ) instrukcja2 ;
     else if ( wyr3 ) instrukcja3 ;
          else if ( wyr4 ) instrukcja4 ;
// dalsza część programu
```

Taką konstrukcję można przeczytać tak:

- Sprawdź **wyr1.**
- Jeśli jest **true** – wykonaj **instr1** i przejdź do dalszej części programu (czyli zakończ pracę w tej całej konstrukcji).
- Jeśli **wyr1** ma wartość **false** (warunek nie jest spełniony), to sprawdź **wyr2** .
- Jeśli **wyr2** jest **true** (jest spełniony), to wykonaj **instr2** i przejdź do dalszej części programu.
- Jeśli zaś **wyr2** jest **false**, to sprawdź **wyr3.** Jeśli on jest **true** spełniony, to wykonaj **instr3** i przejdź do dalszej części programu.
- Natomiast jeśli **wyr3** nie jest spełniony, sprawdź **wyr4** itd.

Operatory- podstawowe pojęcia

Operatorami są pojawiające się w tekście programu specjalne znaki (np.+,*,%,>=,=, itd.) które są interpretowane jako żądanie wywołania odpowiednich funkcji operujących na danych określonych przez wyrażenia sąsiadujące z danym operatorem — są to **argumenty** (lub operandy) tego operatora.

- Operatory generalnie dzielą się na **jedno**- i **dwuargumentowe**.
- Zapis operatorów dwuargumentowych jest **infiksowy**, czyli operator stawiany jest pomiędzy swoimi argumentami.
- Z kolei zapis operatorów jednoargumentowych, z dwoma wyjątkami, jest **prefiksowy** (przedrostkowy), czyli operator stawiamy przed argumentem.
- Istnieje w C/C++ również jeden operator trzyargumentowy: (**operator warunkowy ? :**).

Priorytety i wiązanie

Z samego zapisu infiksowego nie wynika jeszcze kolejność wykonania działań w złożonych wyrażeniach. Np. w wyrażeniu

```
a + b / c
```

zmienna b może być traktowana jako prawy argument operatora dodawania albo lewy argument operatora dzielenia.

Aby uniknąć tego rodzaju wieloznaczności, wprowadzono pojęcie **priorytetu** operatorów.

Operatory zostały podzielone względem priorytetów się na 18 grup, w ramach których priorytety są jednakowe.

Operatory- podstawowe pojęcia - priorytety i wiązanie

- W sytuacjach, jak opisana powyżej, najpierw zostanie wykonana operacja opisywana przez ten z tych dwóch operatorów który ma wyższy priorytet.
- Jeśli natomiast oba mają ten sam priorytet, kolejność wykonywania operacji będzie określona ich **wiązaniem**: od lewej do prawej dla operatorów o **wiązaniu lewym** i od prawej do lewej dla operatorów o **wiązaniu prawym**.
- Aby ta reguła nie prowadziła do sprzeczności, operatory o takim samym priorytecie mają taki sam kierunek wiązania. A zatem w wyrażeniu

```
a + b / c
```

najpierw zostanie wykonane dzielenie (*grupa 5*), gdyż ma wyższy priorytet od dodawania (*grupa 6*).

Operatory- podstawowe pojęcia - priorytety i wiązanie

- Ale w wyrażeniu

  ```
  a + b - c
  ```

  najpierw zostanie wykonane dodawanie, gdyż ma ten sam priorytet co odejmowanie, a oba operatory mają wiązanie lewe (*grupa 6*).
- Dla operatora przypisania wiązanie jest prawe (*grupa 16*). Tak więc w wyrażeniu `a = b = c` przypisanie `b = c` wykonane zostanie najpierw, a jego wynik (wartość `b` po tej operacji) przypisany będzie do zmiennej `a`.
- Jeśli nie jesteśmy pewni jaki jest priorytet operatorów możemy dla pewności w tworzonych wyrażeniach używać nawiasów okrągłych!
- W dalszej części wiązania są lewe, chyba, że zaznaczono inaczej

Operatory arytmetyczne

Operatory arytmetyczne:

\+ dodawania (*grupa 6*),

– odejmowania (*grupa 6*),

* mnożenia (*grupa* 5),

/ dzielenia (*grupa 5*)(rzeczywistego lub
całkowitego dla obu argumentów całkowitych)

Przykłady wyrażeń, w których występują te operatory:

```
a = b + c; // dodawanie
a = b - c; // odejmowanie
a = b * c; // mnożenie
a = b / c; // dzielenie
```

Operatory te wykonują działania na dwóch obiektach (stojących po obu stronach symbolu operatora) i dlatego nazywamy je ***dwuargumentowymi***.

Operatory arytmetyczne

Operator % - reszta z dzielenia (modulo)

Dwuargumentowym operatorem jest także operator reszty z dzielenia % (*grupa 5*).

Np.:

```
int x = 8, n = 5;
cout << "reszta = " << (x % n) << endl;
cout << "iloraz calkowity = " << (x / n) <<endl;
```

na ekranie pojawi się:

```
reszta = 3
iloraz calkowity = 1
```

ponieważ 8 dzielone przez 5 daje resztę z dzielenia 3, a iloraz jest równy 1, bo 5 raz mieści się w 8, 8=1*5+3 .

Operator ten stosuje się tylko dla argumentów całkowitych.

Jednoargumentowe operatory + i – *(grupa 3, wiązanie prawe)*

Jednoargumentowy operator + (plus) właściwie nie robi nic, natomiast **jednoargumentowy operator – (minus)** zamienia wartość danego wyrażenia na liczbę przeciwną.

```
int i = 5;

cout << "i = " << i << ", -i = " << -i << endl;

cout << -(-(-(i+1)));
```

W rezultacie wykonania tego fragmentu na ekranie pojawi się:

```
i = 5, -i =  -5

-6
```

Operatory logiczne

Operatory (*grupa 8*)

< mniejszy niż

<= mniejszy lub równy

> większy niż

>= większy lub równy

są operatorami relacji.

Następnymi operatorami tego typu są operatory (*grupa 9*)

== jest równy

!= jest różny od

Operator **==** to dwa stojące obok siebie znaki = (bez spacji).

Obliczanie wartości wyrażenia logicznego odbywa się w ten sposób, że wynik „prawda” daje rezultat **`true (1)`**, a wynik „fałsz” **`false (0)`**.

Operatory logiczne

Częstym błędem jest omyłkowe postawienie tylko jednego (operator przypisania) zamiast dwóch znaków ==, a wyrażenie przypisania = ma wartość równą wartości będącej przedmiotem przypisania:

```
int a = 5, b = 100;
cout <<"Dane sa: a= " << a << " b= " << b << endl;
if(a = b) //pomyłkowo przypisanie zamiast a==b
  cout << "Zachodzi rownosc"<< endl;
else
  cout << "Nie zachodzi rownosc" << endl;
cout << "Sprawdzam ze: a= " << a << " b= " << b;
```

Wynik:

```
Dane sa: a= 5 b= 100
Zachodzi rownosc
Sprawdzam ze: a= 100 b= 100
```

Operatory sumy logicznej || oraz iloczynu logicznego &&

Operatory :

|| – suma logiczna, czyli operacja logiczna LUB (alternatywa) (*14*)

&& – iloczyn logiczny, czyli operacja logiczna I (koniunkcja)(*gr. 13*)

Standard C++11 dopuszcza również ich „tekstowe” wersje:

- zamiast || możemy pisać (małymi literami) słowo kluczowe **or**, np. `( (x <4) or y)`,
- a zamiast && możemy pisać słowo kluczowe **and**, np.`(q and p)`.

Przykład alternatywy:

```
int k = 2;
if( (k == 2) || (k == 10) ) // alternatywa
{
   cout << "Hurra! k jest równe 2 lub 10 ! ";
}
```

Operatory sumy logicznej || oraz iloczynu logicznego &&

Przykład na koniunkcję:

```
int m = 22, k =77;
if( (m > 10) && (k > 0) ) // koniunkcja
{
  cout << "Hura! m jest wieksze od 10 i "
       <<"równoczesnie k jest wieksze od zera!\n";
}
```

Wyrażenia logiczne tego typu obliczane są od lewej do prawej.

Przy czym ich obliczanie jest optymalizowane tj. **kompilator oblicza wartość wyrażenia, dopóki nie ma pewności, jaki będzie ostateczny wynik**.

Operatory sumy logicznej || oraz iloczynu logicznego &&

Oznacza to, że w wyrażeniu:

```
(a == 0) && (m == 5) && (x > 32)
```

kompilator obliczać będzie od lewej do prawej i jeśli pierwszy człon koniunkcji nie będzie prawdziwy, dalsze obliczanie zostanie przerwane.

Podobnie jest w przypadku alternatywy. Jeżeli w wyrażeniu:

```
if( i || (k > 4) )
{
   // ........
}
```

pierwszy człon alternatywy (zmienna `i`) jest różny od `0` (czyli „prawda”), to dalsze obliczenia nie są już konieczne. Już w tym momencie wiadomo, że alternatywa ta jest spełniona.

Operator negacji ! (*grupa 3, wiązanie prawe*)

Operator negacji `!` jest operatorem jednoargumentowym.

Jego argument stoi po jego prawej stronie, np.:

```
!m
```

Powyższe wyrażenie ma wartość **`true(1)`**, dla `m` równego `0`.

```
int m = 0;
if(!m) cout << "Uwaga, bo m jest równe zero\n";
bool gotowe = false;
if(! gotowe) cout << "jeszcze nie gotowe !\n";
```

Standard C++11 dopuszcza również „tekstową” wersję tego operatora. To znaczy zamiast `!` możemy pisać (małymi literami) słowo kluczowe **`not`**, np. `if(not m)...`

if-else — przykład – rok przestępny

Program sprawdzający czy podany przez użytkownika rok jest przestępny(tj. gdy jest podzielny przez 4 i nie jest podzielny przez 100, chyba że dzieli się przez 400)

```
#include <iostream>
using namespace std;

int main()
{
    int r;
    cout<<"Podaj rok: ";
    cin>>r;

    if((r%4 == 0 && r%100 != 0) || r%400 == 0)
       cout<<"Rok "<< r <<" jest przestępny."<<endl;
    else
       cout<<"Rok "<< r <<"nie jest przestępny."<<endl;
    return 0;
}
```

if-else – przykład – rok przestępny- 2 sposób

```
#include <iostream>
using namespace std;

int main()
{
  int r;
  cout<<"Podaj rok: ";
  cin>>r;
  if(r%400 == 0) //wykorzystanie warunków prostych
    cout<<"Rok "<<r<<" jest przestępny.";
  else if(r%100 == 0)
         cout<<"Rok "<<r<<" nie jest przestępny.";
       else if(rok%4 == 0)
              cout<<"Rok "<<r<<" jest przestępny.";
            else
              cout<<"Rok "<<r<<" nie jest przestępny.";
  return 0;
}
```

if-else –wybór wielowariantowy - przykład

Napisz program, który oblicza ocenę w zależności od liczby punktów zdobytych na sprawdzianie. Obowiązuje następująca punktacja:
0 -50 pkt – dwójka; 51-70 pkt – trójka; 71-90 pkt – czwórka;
90-100 pkt – piątka.
Liczbę punków wczytujemy z klawiatury. W przypadku podania błędnej wartości(za dużo lub za mało) ocena =-1.

```
int main ()
{
   int pkt, ocena;
   cout<<"Podaj liczbe punktow: "<<endl;
   cin>>pkt;
   if ((pkt>=0) && (pkt<=50)) ocena =2;
   else if ((pkt>=51) && (pkt<=70)) ocena=3;
        else if ((pkt>=71) && (pkt<=90)) ocena=4;
             else if ((pkt>=91) && (pkt<=100)) ocena =5;
                  else ocena=-1;
   if(ocena==-1) cout<<"Bledna liczba punktów"<<endl;
   else cout<<"Dostales "<<ocena<<endl;
   return 0;
}
```

if-else –wybór wielowariantowy – przykład wersja 2

Druga, prostsza wersja rozwiązania powyższego zadania:

```
int main ()
{
   int pkt, ocena;
   cout<<"Podaj liczbe punktow: "<<endl;
   cin>>pkt;
   if (pkt < 0) ocena = -1;
   else if (pkt <= 50) ocena = 2;
        else if (pkt <= 70) ocena = 3;
             else if (pkt <= 90) ocena = 4;
                  else if (pkt <= 100)ocena = 5;
                       else ocena = -1 ;

   if(ocena == -1) cout<<"Bledna liczba punktów"<<endl;
   else cout<<"Dostales "<<ocena<<endl;
   return 0;
}
```

Wyrażenie warunkowe *(grupa 15, wiązanie prawe)*

Wyrażenie warunkowe(operator selekcji) jest to wyrażenie, którego wartość może być w pewnej sytuacji taka, a czasem inna.

Jego składnia:

wyr_log ? wyr_tak : wyr_nie

Najpierw obliczana jest wartość wyrażenia **wyr_log** i ewentualnie konwertowana do typu bool.

- Jeśli jest to **true**, obliczane jest wyrażenie **wyr_tak**, a wyrażenie wyr_nie jest ignorowane. Wartością całego wyrażenia jest wartość wyr_tak.
- Jeśli jest to **false**, obliczane jest wyrażenie **wyr_nie**, a wyrażenie wyr_tak jest ignorowane. Wartością całego wyrażenia jest wtedy wartość wyr_nie.

Wyrażenie warunkowe *(grupa 15, wiązanie prawe)*

Przykładowo:

```
(n > 4) ? 15 : 10
```

wyrażenie to w zależności od spełnienia lub niespełnienia warunku (n>4)

przyjmie różną wartość.

- Jeśli warunek jest spełniony, to wartość wyrażenia = 15.
- Jeśli zaś niespełniony, wartość całego wyrażenia wynosi 10.

Wyrażenie warunkowe jest bardzo wygodne, bo może być użyte wewnątrz innych wyrażeń. Na przykład:

```
c = (x > y) ? 5 : 12;
```

do zmiennej `c` zostanie podstawiona liczba `5`, jeśli rzeczywiście `x` jest większe od `y`, a liczba `12`, jeśli `x` nie jest większe od `y`.

Wyrażenie warunkowe *(grupa 15, wiązanie prawe)*

```
int main()
{
  int c;
  cout << "Odpowiedz TAK lub NIE \n"
       << "jesli TAK, napisz 1 \n"
       << "jesli NIE, napisz 0 \n";
  cin >> c;
  cout << "Odpowiedziales: "
       << ( c ? "TAK" : "NIE" ) << ", prawda ? ";
}
```

`c ? "TAK" : "NIE"`– wyrażenie `c` ma wartość `true` jeśli `c!=0`, wówczas całe wyrażenie warunkowe ma wartość `"TAK"`,
jeśli c==0 to warunek jest `false` i całe wyrażenie ma wartość `"NIE"`

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`

- 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";
}
```

Operatory inkrementacji i dekrementacji *(grupa 2)*

Operatory inkrementacji (zwiększenia o jeden) i dekrementacj (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, to:
  - najpierw jest ona zwiększana o 1,
  - następnie ta zwiększona już wartość staje się wartością wyrażenia.
- W przypadku gdy operator inkrementacji stoi za zmienną, 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 otrzymyjemy:

```
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**.