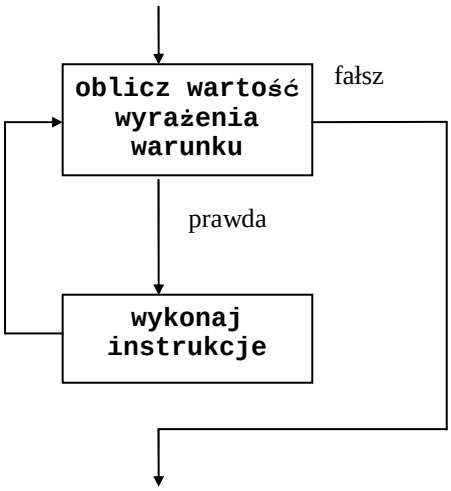
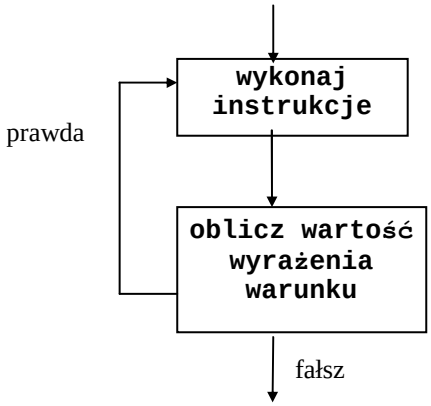
Instrukcje powtarzania

- Instrukcje powtarzania (iteracyjne) pozwalają wielokrotnie wykonać jeden ustalony ciąg instrukcji.
- Instrukcje te tworzą pętle (ang. *loop*).
- Pętle wykonują cyklicznie pewną część kodu programu dopóty, dopóki jakiś określony warunek jest prawdziwy.
- Taki warunek nazywany jest *warunkiem sterowania pętlą*.
- Instrukcje tworzenia pętli to:
    - `while` - warunek powtarzania jest sprawdzany *przed* wykonaniem treści pętli
    - `do-while` - warunek powtarzania jest sprawdzany *po* wykonaniu treści pętli
    - `for` - warunek powtarzania jest sprawdzany *przed* wykonaniem treści pętli

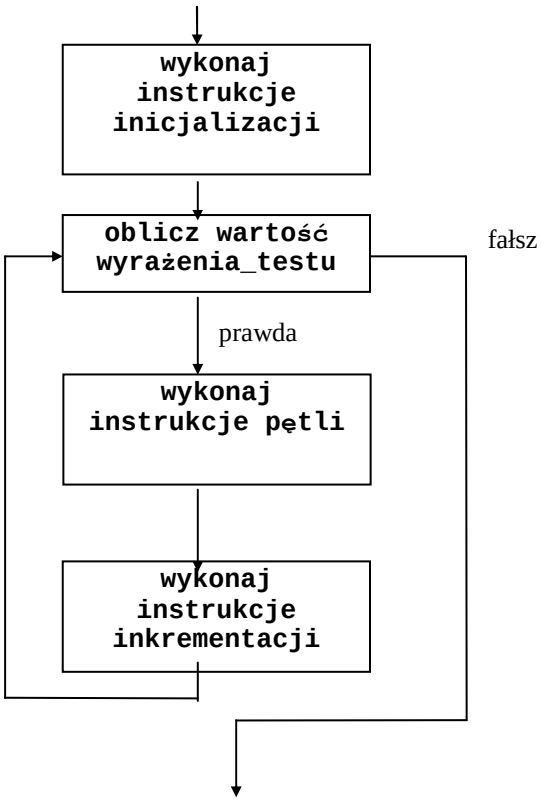
```
while (warunek)
  instrukcja
```




```
do
  instrukcja
while (warunek);
```




```
for (inicjalizacja; warunek; inkrementacja) instrukcja;
```

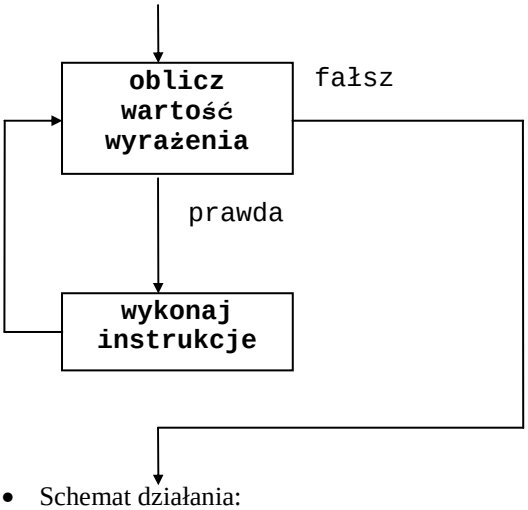


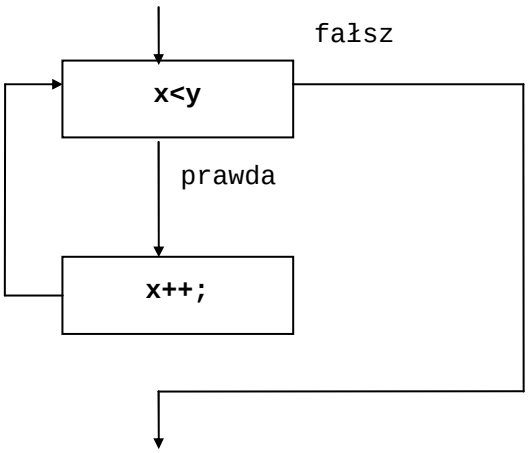
Pętla `while`

- Zasada działania:
  - Powtarzaj instrukcje, dopóki wyrażenie warunkowe jest prawdą.
  - Jeśli już za pierwszym razem warunek nie jest spełniony, to instrukcja związana z `while` nigdy nie będzie wykonana.

```
while (warunek)
  instrukcja
```

```
// dopóki x jest mniejsze od y,
// zwiększaj x o 1
while (x<y)
  x++;
```





- Schemat działania:
  1. Oblicz wyrażenie warunkowe.
  2. Jeśli jego wartością jest `false`, przejdź do instrukcji następującej po pętli. Jeśli zaś `true`, wykonaj instrukcje objęte pętlą.
  3. Wróć na do kroku 1.

- **Przykład 1.**
  Pętla `while` powtarza związane z nią instrukcje tak długo, jak długo warunek kontrolujący wykonywanie pętli jest prawdziwy. Pozwala implementować pętle ogólne.

  A. Dane są dwie liczby całkowite dodatnie *m* i *n*. Chcemy obliczyć ich największy wspólny dzielnik *k*.

  Algorytm klasyczny Euklidesa:
  Dane: dwie liczby naturalne *m* i *n*, $m \leq n$.
  Wynik: *NWD(m,n)* - największy wspólny dzielnik liczb *m* i *n*
  Krok 1: Jeśli *m=0*, to *n* jest szukanym dzielnikiem. Zakończ algorytm.
  Krok 2: *r:=n mod m*, *n=m*, *m=r*. Wróć do kroku 1.

```
#include <iostream.h>
int main ()
{
  int m,n,r;
  cin >> m >> n;  // zakładamy, że m i n > 0
  while (m) // skrót dla while( m != 0)
  {
    r = n % m;
    n = m;
    m = r;
}
  cout << n;
  return 0;
}
```

- B. Chcemy obliczyć sumę ciągu liczb wprowadzanych z klawiatury, dodatkowo chcemy wiedzieć ile liczb zostało wprowadzonych. Koniec wprowadzania jest sygnalizowany znacznikiem końca pliku.

```
int x, licznik=0, suma=0;
while (cin >> x)
{
  ++licznik;   // licznik określa ile liczb wprowadzono
  suma += x;   // suma zawiera sumę wprowadzonych liczb
}
```

  Komentarz:
  - Pętla `while` będzie wykonywana dopóki warunek (`cin >> x`) będzie spełniony. (Dokładne działanie konstrukcji (`cin >> x`) zostanie dopiero wyjaśnione na wykładzie „*Programowanie obiektowe*”). W uproszczeniu: dla obiektu `cin` przeciążony jest operator `()` tak, aby za pomocą warunku (`cin>>x`) można było badać stan strumienia wejściowego, czyli sprawdzać, czy ostatnia próba czytania zakończyła się powodzeniem. Czytanie zakończy się niepowodzeniem wtedy kiedy:
    - napotkany zostanie znacznik końca pliku (można go wygenerować z klawiatury: w DOSie za pomocą klawiszy Ctrl-z, w Unixie klawisze Ctrl-d),
    - próbowano wczytać daną, która jest niezgodna z typem zmiennej, w której ma być umieszczona (na przykład zamiast liczby typu `int` wpiszemy literę),
    - wystąpiło uszkodzenie sprzętowe urządzenia wejścia.

    Stan strumienia wejściowego jest pamiętany dopóki program za pomocą odpowiedniej funkcji (`cin.clear()`) nie przywróci go do stanu umożliwiającego ponowne wczytywanie. Do tego momentu program nie będzie w stanie wczytywać nowych danych z tego strumienia. (Czyli każda nowa próba wczytania nawet poprawnych danych zakończona zostanie niepowodzeniem).

- C. Chcemy obliczyć sumę ciągu liczb dodatnich wprowadzanych z klawiatury, dodatkowo chcemy wiedzieć ile liczb zostało wprowadzonych. Koniec wprowadzania jest sygnalizowany liczbą ujemną lub znakiem końca pliku.

```
int x, licznik=0, suma=0;
while ( (cin >> x) && (x > 0) )
{
  ++licznik;   // licznik określa ile liczb wprowadzono
  suma += x;   // suma zawiera sumę wprowadzonych liczb
}
```

- D. Chcemy zliczyć samogłoski we wprowadzanym tekście. Koniec wprowadzania jest sygnalizowany znacznikiem końca pliku. (Nie uwzględniamy polskich liter).

```
#include <iostream.h>
#include <ctype.h> // dla isalpha()

int main() {
  char zn;
  int liczbaSamoglosek=0, liczbaSpolglosek=0;
  while (cin >> zn)
  {
    switch (zn)
    {
      case 'a' :  case 'A' :
      case 'e' :  case 'E' :
      case 'i' :  case 'I' :
      case 'o' :  case 'O' :
      case 'u' :  case 'U' :
                 ++liczbaSamoglosek;
                 break;
      default  :
                 if (isalpha(zn)) ++liczbaSpolglosek;
                 break;
    }
  }
  cout << "Liczba samoglosek: " << liczbaSamoglosek << endl;
  cout << "Liczba spolglosek: " << liczbaSpolglosek << endl;
  return 0;
}
```

- **Przykład 2.**
  Pętla `while` jest często używana do wczytywania danych wtedy, kiedy nie wiemy z góry ile razy będzie powtórzona. Można wtedy zbudować tzw. pętlę z wartownikiem - czyta ona i przetwarza dane aż do napotkania szczególnego niedozwolonego elementu powodującego zakończenie pętli. Element ten nazywany jest wartownikiem. Nie jest on przetwarzany.
  Pętla z wartownikiem zazwyczaj wymaga wczytania danej przed pierwszym wykonaniem testu. Pseudokod pętli z wartownikiem ma postać:

```
wczytaj daną
while dana nie jest wartownikiem {
  przetwórz daną
  wczytaj daną
}
```

  A. Chcemy obliczyć sumę ciągu liczb zakończonego liczbą 0. Wartownikiem w tym przypadku będzie liczba 0.

```
#include <iostream.h>
int main() {
  int liczba, suma=0;
  cout << "Wpisz liczbe ('0' - koniec) ";
  cin >> liczba;
  while (liczba != 0)
  {
    suma += liczba;
    cout << "Wpisz liczbe ('0' - koniec) ";
    cin >> liczba;
  }
  cout << "Suma: " << suma << endl;
  return 0;
}
```

B. Chcemy wczytywać tekst z klawiatury znak po znaku i wyświetlać go na ekranie do momentu wprowadzania znaku końca pliku. Wartownikiem w tym przypadku będzie znacznik końca pliku EOF.

```
#include <iostream.h>
int main() {
  int znak;
  znak=cin.get();
  while (znak != EOF) {
    cout.put(znak);
    znak=cin.get();
  }
  return 0;
}
```

Komentarz:

- Nie możemy zastosować konstrukcji `cin >> znak`, ponieważ operator `>>` pomija na przykład spacje. Musimy użyć funkcji, która wczytuje jeden znak. Taką funkcją jest `get()` bez argumentu. Pobiera ona jeden znak ze strumienia wejściowego i zwraca go jako wartość funkcji. Jeśli napotka w strumieniu koniec pliku, zwraca znacznik końca pliku EOF. Ponieważ funkcja ta dotyczy strumienia `cin`, który jest obiektem specjalnego typu, musimy użyć składni `obiekt.funkcja()`.
- Stała EOF symbolizuje znacznik końca pliku i jest zdefiniowana w pliku nagłówkowym `iostream.h`. Aby odróżnić znacznik od elementów zbioru znaków, często reprezentuje się go za pomocą liczby –1 (znak jest reprezentowany przez kod ASCII, a więc przez liczbę dodatnią). Zmienną, która przyjmuje wynik funkcji `get()` trzeba zatem zdeklarować jako int, tak aby mogła przyjmować zarówno wartości znaków jak i wartość stałej `EOF`.
- Odpowiednikiem funkcji `get()` dla strumienia wyjściowego `cout` jest funkcja `put()`. Wstawia ona jeden znak (swój argument) do strumienia wyjściowego.

A. Chcemy zliczyć spacje, tabulacje i znaki nowej linii w tekście.

```
#include <iostream.h>
int main() {
  int zn;
  int l_sp=0, l_tab=0, l_nw=0;
  while ((zn=cin.get()) != EOF)  {
    switch (zn)
    {
      case ' '  :  l_sp++; break;
      case '\t' :  l_tab++; break;
      case '\n' :  l_nw++; break;
    }
  };
  cout << "Podsumowanie" << endl;
  cout << "Liczba spacji:         " << l_sp << endl;
  cout << "Liczba tabulacji:      " << l_tab << endl;
  cout << "Liczba nowych wierszy: " << l_nw << endl;
  return 0;
}
```

Przykład 3
Jeśli zawczasu wiemy, ile razy pętla będzie powtarzana możemy utworzyć pętlę z licznikiem - przetwarza ona dane tyle razy, ile wskazuje licznik.

```
// Załóżmy, że licznik i zmienia się od a do b
i=a;
while (i<=b) {
  // przetwórz i-ty element
  i++;
}
```

Komentarz:

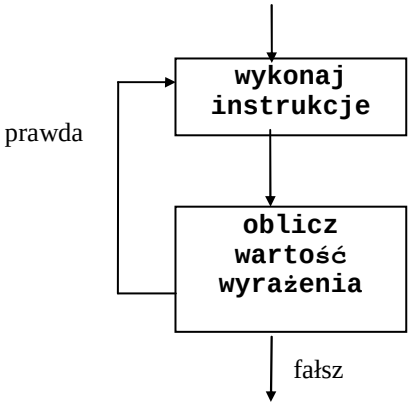
- tego typu pętlę częściej buduje się z wykorzystaniem konstrukcji `for`.

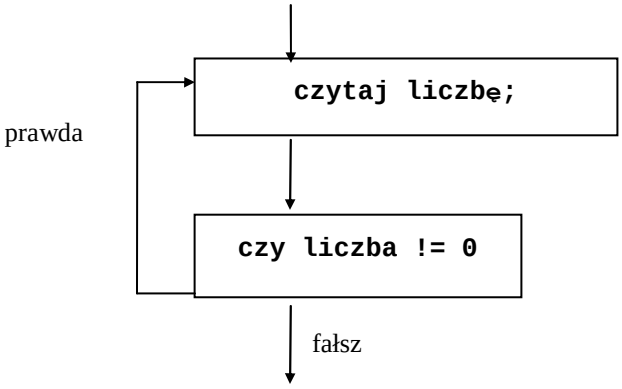
Pętla `do.. while`

- Zasada działania:
  - Wykonaj instrukcje jeden raz. Sprawdź wartość wyrażenia warunkowego. Powtarzaj instrukcje, dopóki wyrażenie to jest prawdą.
  - Jeśli już za pierwszym razem warunek nie jest spełniony, to instrukcja związana z `do .. while` będzie wykonana dokładnie jeden raz.

```
do
  instrukcja
while (warunek);
```

```
do
{
  cin >> liczba;
}
while (liczba != 0);
```





Schemat działania:

1. Wykonaj instrukcje pętli.
2. Oblicz wyrażenie warunkowe. Jeśli jego wartością jest `false`, przejdź do instrukcji następującej po pętli. Jeśli zaś `true`, wróć do kroku 1.

Przykład 1. Obliczyć sumę liczb od 1 do 10.

```
licznik=1;
do
{
   cout << licznik;
   ++licznik;
}
while (licznik<=10);
```

Przykład 2. Wprowadzić liczbę z zakresu (2,5).

```
int min=2, max=5,x;
do
{
  cout << "Wpisz liczbę z zakresu od " << min << " do " << max <<": ";
  cin >> x;
}
while (x<min || x > max);
```

Przykład 3. Chcemy na przykład napisać program interaktywny, który po wykonaniu części obliczeń będzie pytał kontynuować pracę:

```
cout << "Kontynuowac (T/N): ";
do
{
  cin >> z;
  z=toupper(z);
}
while (z != 'T' && z != 'N');
```

Przykład 4. Pętlę `do..while` często wykorzystuje się wtedy, kiedy sterowanie pętlą jest ustalane wewnątrz treści pętli. Chcemy na przykład napisać program interaktywny, który po wykonaniu obliczeń będzie pytał czy powtórzyć je dla innych danych:

```
do
{
  // instrukcje
} while (czyPowtorzyc())  // czyPowtorzyc jest funkcją, która zwraca true
                          // jeśli użytkownik zdecydował się kontynuować,
                          // false w przeciwnym wypadku
```

To samo za pomocą pętli `while`:

```
// trzeba podać pierwszą wartość, aby rozpocząć pętlę
bool powtorzyc=true;
// teraz można rozpocząć obliczenia
while (powtorzyc) {
  // instrukcje
  powtórzyć=czyPowtorzyc();
}
```

Przykład 5. Sumowanie liczb. Liczby wczytywane są do wpisania 0.

```
#include <iostream.h>
int main() {
  int liczba, suma=0;
  do
  {
    cout << "Wpisz liczbe ('0' - koniec) ";
    cin >> liczba;
    suma += liczba;
  }
  while (liczba != 0);
  cout << "Suma: " << suma << endl;
  return 0;
}
```

Przykład 6. Proste menu.

```
do
{
  wybor=cin.get();
  switch (wybor)
  {
    case '1': dodaj_rekord();
              break;
    case '2': usun_rekord();
              break;
    case '3': szukaj_rekord();
              break;
  }
}
while (wybor!='1' && wybor !='2' && wybor !='3');
```

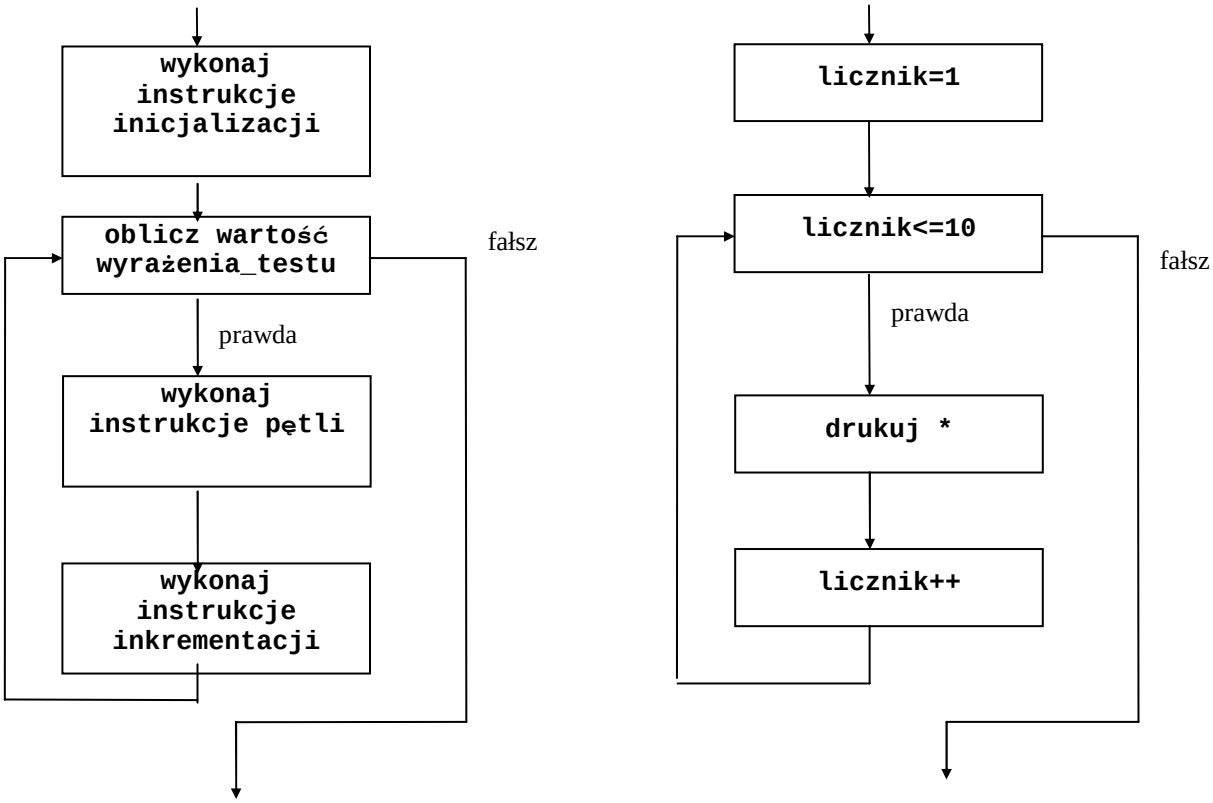
Pętla for

```
for (inicjalizacja; warunek; inkrementacja)
  instrukcja;
```

gdzie:

inicjalizacja	zwykle instrukcja przypisania wartości początkowej zmiennej sterującej pętli (licznika); jest wykonywana raz, przed rozpoczęciem powtarzania
warunek	wyrażenie określające warunek powtarzania; sprawdzane jest przed każdym kolejnym powtórzeniem pętli
inkrementacja	zwykle instrukcja modyfikująca zmienną sterującą pętli po zakończeniu każdego jej przebiegu (powtórzenia); jest wykonywana po każdym powtórzeniu pętli ale przed sprawdzeniem warunku powtarzania

```
// wydrukuj 10 gwiazdek
for (licznik=1; licznik<=10; licznik++)
  cout << "*";
cout << endl;
```



Schemat działania:

1. Oblicz wyrażenie inicjujące pętli.
2. Oblicz wyrażenie warunkowe pętli (sterujące powtarzaniem).
3. Jeśli jego wartością jest `false`, przejdź do instrukcji następującej po pętli. Jeśli jego wartość jest `true`, wykonaj instrukcje pętli i następnie wyrażenie końcowe inkrementacji. Wróć do kroku 2 - sprawdzania wyrażenia warunkowego pętli.

Przykład 1. Typowe użycie pętli `for` to pętla z licznikiem: przetwarzaj dane tyle razy, ile wskazuje licznik.

```
for (licznik=1; licznik < ILE; licznik++)
{
  // instrukcje
}
```

Równoważny zapis z użyciem `while`:

```
licznik=1;
while (licznik < ILE) {
   // instrukcje
   licznik++;
}
```

A. Chcemy obliczyć kwadraty liczb od 1 do 10

```
for (x=1; x<=10; x++)
{
   z = x*x;
   cout << "Kwadrat liczby " << x << " = " << z << endl;
}
```

B. Chcemy obliczyć sumę pierwszych 5 liczb naturalnych.

```
// Rozwiązanie 1: za pomocą pętli for
suma=0;
for (i=1;i<=5;i++)
  suma += i;

// Rozwiązanie 2: za pomocą pętli while
suma=0;
i=1;
while (i<=5)
{
  suma += i;  /* zamiast suma = suma+i; */
  i++;        /* zamiast i = i+1; */
}

// Rozwiązanie 3: za pomocą pętli do-while
suma=0;
i=1;
do
{
  suma += i;
  i++;
}
while (i<=5);
```

C. Licznik określający liczbę powtórzeń może być budowany na różne sposoby.

```
// zmienna znak będzie przyjmować wartości od 'a' do 'z'
for (char znak='a'; znak <= 'z' ; znak++)
{ // instrukcje }

// x przyjmuje wartości od 1.5 do 9.75 z krokiem 0.25
for (double x=1.5; x<10; x += 0.25)
{ // instrukcje }

// wykladnik przyjmuje wartości od 1 do n
for (potega=1.0, wykladnik=1; wykladnik<=n; ++wykladnik)
  potega *= n;
```

Przykład 2. Przy wyborze pętli warto jako jedno z kryteriów przyjąć czytelność programu. Przykład użycia pętli `for` w przypadku, gdy bardziej naturalne jest użycie pętli `while` :

```
for (bool znaleziono=false; !znaleziono;)
{
  // instrukcje
  znaleziono=szukaj();
  // instrukcje
}
```

Równoważny zapis z użyciem `while`:

```
bool znaleziono=false;
while (!znaleziono) {
  // instrukcje
  znaleziono=szukaj();
  // instrukcje
}
```

Przykład 3. Napisać program, który sprawdza wyniki testu. Wyniki te przechowywane są w postaci skompresowanej - wektora bitowego. W wektorze odpowiedzi 1 oznacza odpowiedź "tak", zaś 0 - "nie". Program ma utworzyć i wyświetlić wektor z niepoprawnymi odpowiedziami – w wektorze tym 1 oznacza, że nie odpowiedziano prawidłowo.

```
#include <iostream.h>
/* prawidlowe odpowiedzi testu
   nntt nttt tttt nntt
   0011 0111 1111 0011
*/

const unsigned short int ODPOWIEDZ=0x37F3; //wzorzec prawidłowej odpowiedzi
int main() {
  unsigned short int i, wynik_testu=0, zle_odp;
  char c;
  cout << "Wpisz wyniki testu:" << endl;

  // ustaw bity odpowiedzi
  for (i=0; i<16; i++)
  {
    c=cin.get();
    if (c=='t' || c=='T')
      wynik_testu |= (1 << i); // skrót wynik_testu=wynik_testu | (1 << i);
  }

  // wyświetl wektor odpowiedzi
  cout << "Wpisano=" << endl;
  for (i=0x8000;i;i>>=1)   // i=i>>1
    cout <<  ((i & wynik_testu) != 0); // skrót dla
                                       // if (i & wynik_testu) cout <<"1";
                                       // else cout << "0";
  cout << endl;

  // pozostaw bity z odpowiedziami niepoprawnymi
  zle_odp=wynik_testu^ODPOWIEDZ;

  // wyświetl wektor niepoprawnych odpowiedzi
  cout << "Zle odpowiedzi=" << endl;
  for (i=0x8000;i;i=i>>1)
    cout <<  ((i & zle_odp) != 0);  // skrót dla
                                //   if (i & zle_odp) cout <<"1 ";
                                //   else cout << "0 ";
  cout << endl;
  return 0;
}
```

Przykład 4. Pętle zagnieżdżone

A. Wydrukować tabliczkę mnożenia dla liczb od 1 do 5

```
// wersja 1a
#include <iostream.h>
int main() {
  int i,j;
  const int max=5;
  for (i=1;i<=max;i++) {
    for (j=1;j<=max;j++)
      cout << i*j << ' ';
    cout << endl;
  }
  return 0;
}
```

Wynik działania programu:

```
1 2 3 4 5
2 4 6 8 10
3 6 9 12 15
4 8 12 16 20
5 10 15 20 25
```

```
// wersja 1a
#include <iostream.h>
int main() {
  int i,j;
  const int max=5;
  for (i=1;i<=max;i++) {
    for (j=1;j<=max;j++)
      cout << i*j << '\t';
    cout << endl;
  }
  return 0;
}
```

Wynik działania programu:

```
1   2   3   4   5
2   4   6   8   10
3   6   9   12  15
4   8   12  16  20
5   10  15  20  25
```

B. Chcemy wydrukować tabliczkę mnożenia, ale dodatkowo nadać wynikom postać:

```
     1  2  3  4  5
   ---------------
 1|  1  2  3  4  5
 2|  2  4  6  8 10
 3|  3  6  9 12 15
 4|  4  8 12 16 20
 5|  5 10 15 20 25
```

```
// wersja 2
#include <iostream.h>
#include <iomanip.h>   // dla setw()
int main() {
  int i,j;
  const int max=5;  // tabliczka mnożenia do 5
  const int w=4;    // wynik umieszczany jest w polu 4 znakowym
  cout << setw(w+1) << ' ';
  for (i=1;i<=max;i++)
    cout << setw(w) << i;
  cout << endl;
  cout << setw(w+1) << ' ';
  for (i=1;i<=max*w;i++)
    cout << '-';
  cout << endl;
  for (i=1;i<=max;i++) {
    cout << setw(w) << i <<'|';
    for (j=1;j<=max;j++)
      cout << setw(w) << i*j;
    cout << endl;
  }
  return 0;
}
```

Komentarz:
Do zmiany formatowania strumienia wyjściowego służą manipulatory. Manipulator `setw()` pozwala ustalić szerokość wyjściowej reprezentacji liczby lub napisu. Jako argument podaje mu się wymaganą szerokość pola w znakach.