Wykład 2

- Instrukcje języka C

Instytut Informatyki

dr inż. Marcin Radom

Instrukcje

- **Instrukcja** to definicja obliczenia i określenie sposobu wykonania tego obliczenia.
- W języku C wyrażenie takie jak *x = 0* czy *float a = funkcja(16, 20, 67.35)* staje się **instrukcją** jeśli kończy się średnikiem ; np.:

```
float a = funkcja(16, 20, 67.35);
```

- **Program** rozumiany jest jako ciąg instrukcji wykonywanych kolejno od pierwszej do ostatniej.
- **Przekład instrukcji** to fragment programu w języku wewnętrznym realizujący obliczenie zdefiniowane tą instrukcją.
- **Przykładowe instrukcje proste (pojedyncze instrukcje)**

```
int   m,  n = 1;
;                       // instrukcja pusta
m = n * n - 1;          // zmiana wartości m
n++;                    // zmiana wartości n
m + n;                  // bez rezultatu
```

Instrukcje

- **Blok instrukcji (instrukcja złożona)** zawarty jest pomiędzy nawiasami klamrowymi, tak aby całość składniowo była równoważna jednej instrukcji:

  { instrukcja_1; instrukcja_2; … }

- Po nawiasie } (zamykającym blok) nie występuje średnik!
- Bloki instrukcji mogą być zagnieżdżone, czyli bloki {} wewnątrz innych bloków {}, itd.
- W ramach bloku można deklarować zmienne.
- **Przykładowe instrukcje złożone:**

```
float p, q;
{
      p = 3.5;
      q = 7.1 + p++ ;
}

{p = q; q = 1;}      // średnik przed } nie może zostać opuszczony
```

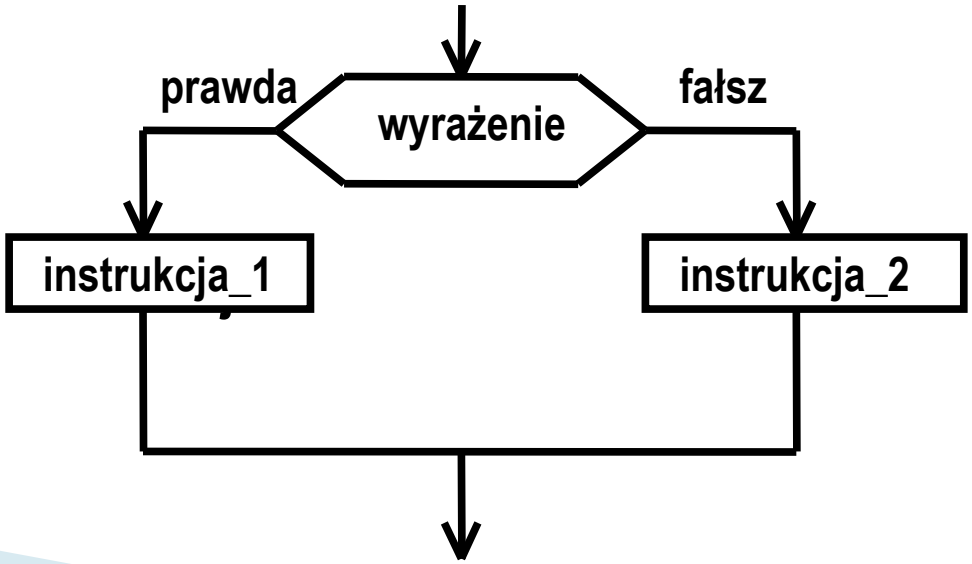
Instrukcja if-else

- Teoria, przykłady

Instrukcja warunkowa if-else

- Stosuje się ją przy podejmowaniu decyzji.
- **Formalna postać:**

```
if (  wyrażenie )
      instrukcja_1
else
      instrukcja_2
```



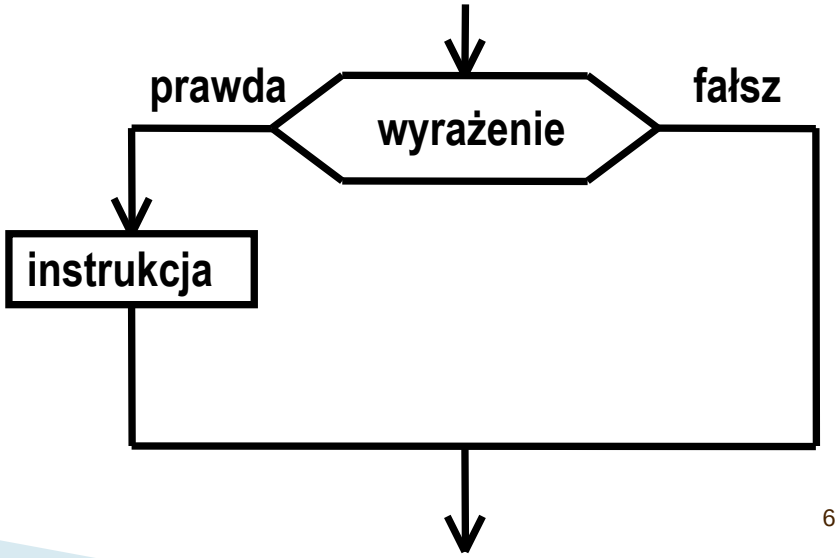
Instrukcja warunkowa if-else

- Najpierw obliczana jest wartość *wyrazenia*. Jeśli wartość *wyrazenia* jest różna od zera, czyli innymi słowy *wyrazenie* jest spełnione/prawdziwe) to zostanie wykonana *instrukcja_1*. Jeśli *wyrazenie* jest fałszywe (tzn. wartość *wyrazenia* jest równa zero) oraz jeśli istnieje część **else** to zostanie wykonana *instrukcja_2.*
- Zgodnie z tą zasadą, nie ma żadnej różnicy pomiędzy (**instrukcja if sprawdza numeryczną wartość wyrażenia**):

```
if (wyrazenie) { /* … */ }   a     if (wyrazenie != 0) { /* … */ }
```

- Przy czym część *else* można pominąć:

```
if (  wyrażenie  ) instrukcja
```



Instrukcja warunkowa if-else

- Instrukcje **if-else** można zagnieżdżać:
  - Ze względu na to, że część **else** nie jest obowiązkowa może wystąpić niejednoznaczność, gdy w ciągu zagnieżdżonych instrukcji **if** jedna z części **else** zostanie pominięta.
  - W takim przypadku instrukcja **else** 'należy' zawsze do ostatniego wyrażenia **if**.
  - Oba wyrażenia powyżej są więc równoważne.

- **Np.** (Choć wcięcie i tabulację ustawiono na poziomie pierwszej instrukcji **if (n>0)**, **else** WCIĄŻ należy do **if (a>b)** ! Taki błąd może być ciężki do wykrycia. Dlatego nie warto oszczędzać na nawiasach.):

```
if (n > 0)
     if (a > b)
               z = a;
else
     z = b;
```

Instrukcja warunkowa if-else

- **Przykłady:**
  - Dla instrukcji warunkowej **if** bez części **else**.

```
long   k, m;
char  flaga;

if (k > m)   flaga = 0;
if (k < m) {
    flaga = 1;
    k = m - k;
}
if (m == 1)   {         // !!!
     if ( k )          // k != 0
     flaga = 2;
     if ( !k )         // k == 0
         flaga = 3;
}
```

Instrukcja warunkowa if-else

- Dla instrukcji warunkowej **if-else**:

```
int   i,  f;
if (i > 5)
     f = 3;
else
     --f ;
/* średnik ; przed else nie może zostać opuszczony */

double   ma,  winien,  saldo,  debet;
if (ma > winien) {
     saldo = ma - winien;
     debet = -1;
} else {
     saldo = -1;
     debet = winien - ma;
}
```

- **Inne:**

```
if   (a)   if   (b)   c;   else   d;
/* jest równoważne*/
if   (a)   {  if   (b)   c;
          else  d;  }
```

Instrukcja warunkowa if-else

```
if   (a)   if   (b)   c;   else  d;  else
if   (e)   f;   else   g;
   /* jest równoważne*/
if (a) {
       if (b)
            c;
       else
            d;
}
else {
       if (e)
             f;
       else
             g;
}
```

- **Przykłady programów:**

```
       void main() { // wymaga załączenia dodatkowo biblioteki math.h
            // deklaracje
            double a, b,                    // parametry
                   G;                       // wynik
            int dobrze = 1;
            // wczytywanie danych
            printf ("\nPodaj wartosc a  ");            // zaproszenie
            scanf ("%lf", &a);                         // wczytywanie
```

Instrukcja warunkowa if-else

```
        printf ("\nPodaj wartosc b  ");          // zaproszenie
        scanf ("%lf", &b);                       // wczytywanie

        // obliczenie  wyniku
        if (a >= b) {
             if ( -b > 0)
                  G = a * a + log(-b);
             else
                  dobrze = 0;
        } else {
             if (b >= 0)
                  G = a - sqrt(b);
             else
                  dobrze = 0;
        }

        // wyprowadzenie wyników
        if (dobrze)
             printf("\nWartosc G wynosi : %.4lf\n\n", G);
        else
             printf("Wyniku nie mozna obliczyc.\n\n");
}
```

Instrukcja warunkowa if-else

```
int main() { // wymaga załączenia dodatkowo biblioteki math.h
     // deklaracje
     double x, y,                // parametry
           F;                    // wynik
     // wczytywanie danych
     printf ("\nPodaj wartosc x  ");        // zaproszenie
     scanf ("%lf", &x);                     // wczytywanie

     printf ("\nPodaj wartosc y  ");        // zaproszenie
     scanf ("%lf",&y);                      // wczytywanie

     // obliczenie  wyniku
     if (x > y)
           F = x * x + y - 1;
     if (x == y)
           F = sin(y) + 2;
     if (x < y)
           F = cos(x) - y + 2;

     // wyprowadzenie wyników
     printf("\nWartosc F wynosi : %.4lf\n\n", F);
     return 0;
}
```

Instrukcja warunkowa – ogólne decyzje wielowariantowe

- **Konstrukcja:**

```
if (wyrażenie)     {
        instrukcja;
} else if (wyrażenie) {
        instrukcja;
} else if (wyrażenie) {
        instrukcja;
} else {
        instrukcja;
}
```

- **Np.:**

```
if (x > y) z = 1;
else if (x < y) z = -1;
else z = 0; // x == y
```

- Taka forma instrukcji **if** pozwala podejmować o wiele bardziej skomplikowane decyzje niż tylko: albo jedno albo drugie.
- W zasadzie nie jest to żadna nowa odmiana **if-else** z poprzednich slajdów, ale raczej jej rozwinięcie.
- Kolejno oblicza się wartości wyrażeń. Pierwsze napotkane wyrażenie prawdziwe spowoduje wykonanie związanej z nim *instrukcji* i zakończenie wykonywania całej konstrukcji.
- Ostatnia część **else** oznacza „żaden z powyższych warunków” i jest wykonywana w sytuacji, w której wszystkie poprzednie wyrażenia były fałszywe.

Instrukcja switch

- Oraz: case, default, break (ta ostatnia gościnnie)

Instrukcja wyboru **switch**

- Instrukcja **switch** służy do podejmowania decyzji, w których sprawdza się, czy wartość pewnego wyrażenia pasuje do jednej z kilku całkowitych stałych wartości i wykonuje odpowiedni skok.

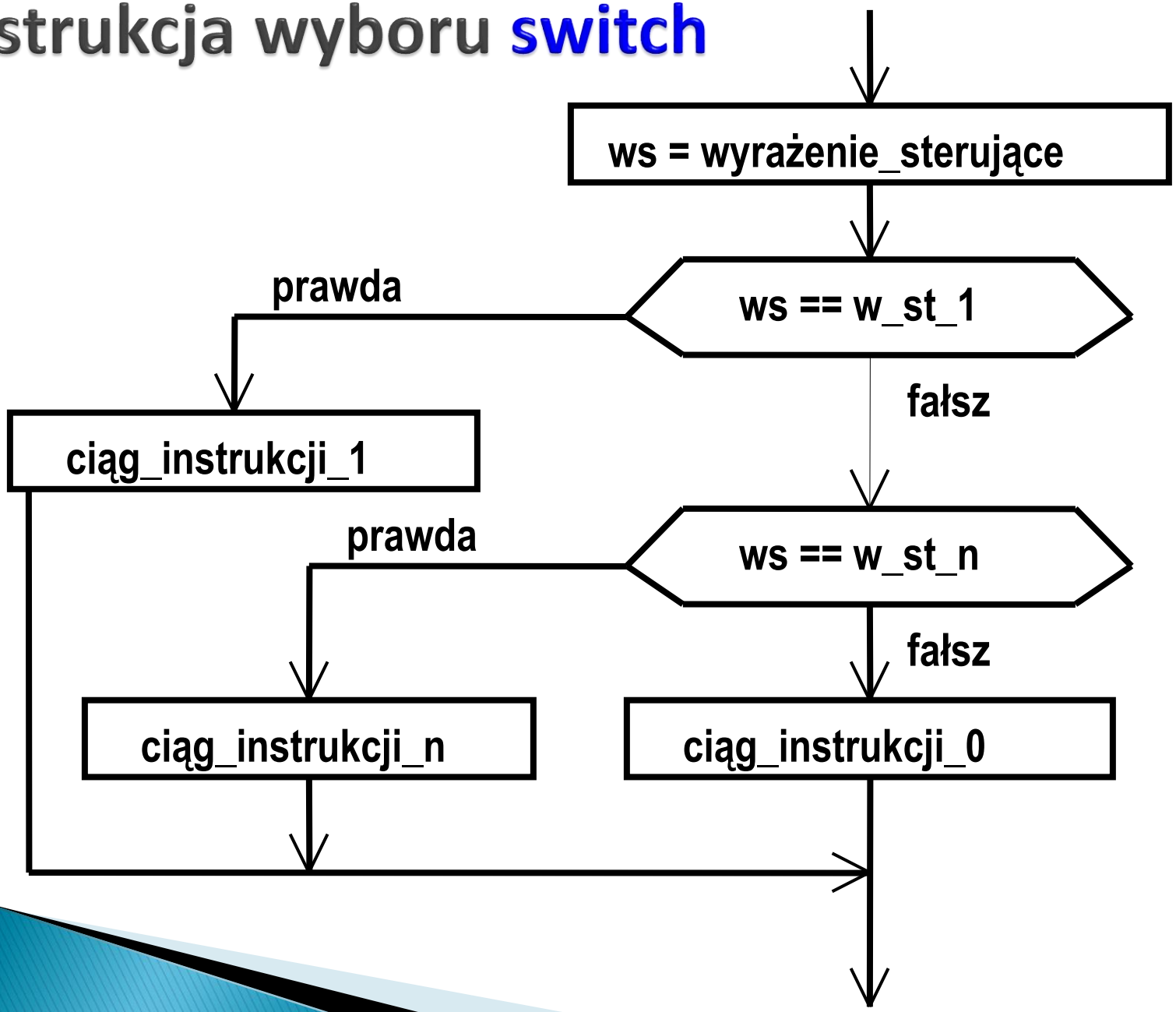
- **Formalna postać:**

```
switch   (   wyrażenie_sterujące   )
{
  case  wyrażenie_stałe_1   :
              ciąg_instrukcji_1
              break;
  . . . . . . . . . . . . . . . . . . . . . . . . . . . .
  case  wyrażenie_stałe_n   :
              ciąg_instrukcji_n
              break;
  default   :  ciąg_instrukcji_0
}
```

Instrukcja wyboru **switch**

- Z każdym wariantem związana jest jedna lub kilka wartości całkowitych lub wyrażeń stałych.
- Jeżeli jeden z przypadków (**case**) jest zgodny z wyrażeniem, wykona się odpowiedni blok instrukcji. Jeśli żadne wyrażenie nie jest zgodne, wtedy wykonywane są instrukcje po klauzuli **default**.
- **default** nie jest obowiązkowe. Jeśli go nie będzie, a żadne wyrażenie nie będzie pasować, wtedy po prostu program zacznie wykonywać polecenia po **switch**.
- Instrukcja **break** powoduje natychmiastowe wyjście z instrukcji **switch**.
- **Po wykonaniu instrukcji związanych z jednym przypadkiem (etykietą) sterowanie przechodzi do następnego przypadku jeśli nie umieszczono instrukcji break.**
- Do dobrego stylu programowania należy wstawianie **break** po ostatniej instrukcji ostatniego przypadku (mimo, że nie jest to konieczne).

Instrukcja wyboru switch



Instrukcja wyboru switch

- **Przykłady:**

```
int   ile_a = 0,  ile_b = 0,  ile_xy = 0,  nieznany = 0;
char zn;
switch  (zn) {
   case  'a' :  ++ile_a;   break;
   case  'b' :  ++ile_b;   break;
   case  'x' :
   case  'y' :  ++ile_xy;  break;
   default   :  ++nieznany;
}
// Przykład różnej interpretacji języka C przez różne kompilatory (VS2012
// akceptuje ten kod i wykonuje go poprawnie, DevC++ nie kompiluje).
// Przykład ten pokazuje, iż stałe w przełączniku mogą być obliczane
// np. za pomocą iloczynów bitowych

int stan, nast;
const int maska = 0x3A;          // const-qualified variable is not a constant expression
                                 // it's a value you cannot modify
switch (stan & maska) {  // error: case label does not reduce to an integer constant
  case   maska & 0x02 :  nast = 0x15; stan = 0x21; break;
  case   maska & 0x30 :  nast = 0x1F; stan = 0x21; break;
  default :   stan = 0;  nast = 0;
}
```

Instrukcja wyboru switch

```
void main() {
      // deklaracje
      int x;              // parametr
      char opcja;

      // wczytywanie danych
      printf ("\nPodaj wartosc x : ");          // zaproszenie
      scanf ("%d", &x);                         // wczytywanie
      printf ("\nWybierz opcje [D, H, X, F] :  ");          // zaproszenie
      fflush(stdin);                                        // oczyszczenie bufora
      scanf ("%c", &opcja);                                 // wczytywanie

      // wyprowadzenie wyniku
      switch (opcja & 0x5F) { // zamiana małych liter ASCII na duże
            case 'D'    :           printf("\n%d\n\n", x); break; // dziesiętnie
            case 'H'    :           printf("\n%x\n\n", x); break; //szesnastkowo, małe litery
            case 'X'    :           printf("\n%X\n\n", x); break; //szesnastkowo, duże litery
            case 'F'    :           printf("\n%.2f\n\n", (float)x); break; // float
            default     :           printf("Zla opcja.");
      }
}
```

Instrukcje pętli: for

- Teoria, przykłady

Pętla for

- **Pętla:**

```
for  ( inicjalizacja ; wyrażenie_graniczne ;  wyrażenie_zliczające  )
            instrukcja_powtarzana
```

- Część inicjująca pętlę (inicjalizacja) wykonuje się raz przed wejściem do właściwej pętli.
- Zakończenie wykonywania pętli następuje gdy wyrażenie graniczne (które jest warunkiem sterującym powtarzaniem pętli i stanowi warunek zakończenia) przyjmie wartość **false** (czyli równą zero)
- Wyrażenia w pętli **for** można pominąć (nawet wszystkie), ale muszą pozostać średniki.
- Wyrażenie graniczne jeśli zostanie pominięte, uznaje się, że jest zawsze spełnione, tj. pętla wykonuje się wtedy w nieskończoność, np.

```
for ( ; ; )
{ /* … */ } //pętla nieskończona
```

Pętla for

- Popularny przykład pętli:

```
for( i = 0; i < n ; i++)   {
    // iteracja po n elementach
}
```

- Wewnątrz pętli można zmieniać warunek końca pętli (wyrażenie graniczne) oraz wartość zmiennej sterującej *i*.

- Po zakończeniu wykonywania zmienna sterująca *i* ma ostatnią wartość przypisaną przez **for** (niezależnie od przyczyny zakończenia pętli).

- Przykład **funkcji zmieniającej ciąg** znaków na liczbę:

```
// każdy krok wykonuje swoje zadanie i pozostawia następnemu jasną sytuację
#include <ctype.h>
int atoi(char s[]) {
        int i, n, sign;                          // znaki białe: \n \t \v \f \r spacja
        for (i = 0; isspace(s[i]); i++)
                        ;                        // pomiń białe znaki (wynik różny od 0)
        sign = (s[i] == '-') ? -1 : 1;
        if (s[i] == '+' || s[i] == '-') i++;     // pomiń symbol znaku
                                                 // isdigit zwraca wartość różną od zera gdy argument jest cyfrą,
        for (n = 0; isdigit(s[i]); i++)          // dla wszystkich cyfr
                        n = 10 * n + (s[i] - '0'); // np. 124 -> 10*0+1, 1*10+2, 12*10+4=124
        return sign * n;
}
```

Pętla for

- **Przykład:**

```
int  s = 0;
for  ( int i = 0;  i <= 9;  ++i)  s += i;
/* inicjalizacja to definicja zmiennej */
```

- **Przykład:**

```
int  i, k = 1525 ;
long m ;
// przecinek w pętli for, wyrażenia obliczane są od lewej do prawej
for  ( i = k, m = 0; i > 0; i -= 3 )
{
   if (i & 1)  ++m ; // zliczana jest ilość indeksów nieparzystych
}
/* inicjalicja to instrukcje przypisania */
```

Pętla for

```
bool dalej = true;
int gdzie;
for (int i = 0; i < N && dalej; ++i)
{
      ........
      if ( .... )
          dalej = false;    //czyli jak zakończyć for bez rozkazu break
      else
          ........
}
// koniec widoczności zmiennej i
```

Pętla for

```
bool dalej = true;
int i,  gdzie = -1;
const int N = 12, Szukany = 333;
int Tab[N] = {0, 1, 333};

for (i = 0; i < N && dalej; ++i)
{
     if ( Tab[i] == Szukany )
          dalej = false;
}
gdzie = i;                    // wartość i == 3
```

Pętla for

```c
#include <string.h>

/* reverse: odwracanie kolejności znaków w s[] */
void reverse(char s[]){
     int c, i, j;

     // przecinek w pętli for, wyrażenia obliczane są od lewej do prawej
     for (i = 0, j = strlen(s)-1; i < j; i++, j--)
     {
          c = s[i];
          s[i] = s[j];
          s[j] = c;
     }
}
```

```
instrukcja      i = 0, j = 9
anstrukcji      i = 1, j = 8
ajstrukcni      i = 2, j = 7
ajctruksni      i = 3, j = 6
ajckrutsni      i = 4, j = 5
ajckurtsni
```

- Przecinki oddzielające argumenty funkcji, zmienne w deklaracjach itp. nie są operatorami i nie gwarantują obliczeń od lewej do prawej.

- *strlen(s)* – podaje długość ciągu znaków s, więc jeśli s ma 7 znaków, musimy odjąć 1 aby uzyskać indeks ostatniego (ponieważ numerowane są od zera!)

Instrukcje pętli: while

- Teoria, przykłady, pętla do – while

Pętla while

- **Pętla:**

```
while  (  wyrażenie_graniczne  )
       instrukcja_powtarzana
```

- Najpierw sprawdzane jest wyrażenie graniczne. Jeśli jest **prawdziwe (tj. różne od zera)** to wykonywana jest treść pętli (instrukcje powtarzane).

- Pętla **while** *(wyrażenie_graniczne)* wykonuje się, dopóki wyrażenie graniczne jest różne od zera. W chwili, gdy wyrażenie graniczne stanie się **fałszywe (tj. równe 0)** nastąpi koniec pętli i zostanie wykonana pierwsza instrukcja po pętli.

- Oczywiście, aby uniknąć pętli nieskończonej, gdzieś wewnątrz { } pętli **while** należy zmodyfikować wyrażenie graniczne, aby w pewnym momencie przestało być spełniane.

Pętla while

- **Przykład:**

```
float   suma = 1573.821,   skladnik = 3.51;
int  licznik = 0;

while (suma > 1E-10)              // 1E-10 to inaczej 1×10-10
{
   suma -= skladnik;
   skladnik *= skladnik;
   ++licznik;
}
// licznik = 4, suma = -21632.44
```

Pętla while

```c
#include <stdio.h>
#include <math.h>

int main() {
	int n = 0, K = 1;

	// wczytywanie danych, aż do podania wartości poprawnej
	while (n < 1){
		printf("Podaj wartosc calkowita n [n > 0] : ");
		scanf("%d", &n);
	}

	while (n > 1){
		K *= (n - 1) * (n - 1) + 1;
		--n; // modyfikacja n, wpływa na wyrażenie graniczne w pętli
	}

	printf("K = %d\n", K);
	return 0;
}
```

Pętla for vs while

- Stosowanie **for** lub **while** często zależy od osobistych preferencji programisty.
- **Przykład:**

```
for (i = 0; i < 10; i++){ /* … */ }
```

jest równoważne:

```
i = 0;
while( i < 10 ){
        // …
        i++;
}
```

- Tam gdzie istnieją proste części inicjujące i zliczające lepiej jest korzystać z instrukcji **for,** ponieważ skupia instrukcje sterujące w jednym miejscu, na szczycie pętli.
- W pętli **while** nie występuje ani część inicjująca, ani część modyfikująca, więc użycie jej jest bardziej naturalne.

Pętla for vs while

```
#include <stdio.h>
#include <math.h>
void main() {
     int n, i;        // zmienna bieżąca
     double a, S;     // parametr
                      // wartość niepoprawna (spełnia warunek pętli while)
     n = -1;
                      // wczytywanie danych, aż do podania wartości poprawnej
     while ( n < 1 )  {
           printf ("\nPodaj wartosc graniczna n (liczba calkowita wieksza od 0) : ");
           scanf ("%d", &n);
     }
     printf ("\nPodaj wartosc parametru a (liczba rzeczywista) : ");
     scanf ("%lf", &a);
                      // obliczenie  sumy
     S = 0.0;         // wartość neutralna
                      // sumowanie odbywa się n razy
     for ( i = 1 ; i <= n ; ++i )
           S += ( a * pow(i, 3.0) - 7 ) / ( i * i + 1 );

                      // wyprowadzenie wyników
     printf("\nWartosc sumy wynosi : %.4lf\n\n", S);
}
```

Pętla do-while

- **Pętla:**

```
do   instrukcja_powtarzana
while   (  wyrażenie_graniczne  ) ;
```

- Pętle **`while`** oraz **`for`** sprawdzają warunek (wyrażenie graniczne) przed wykonaniem pierwszej iteracji. Możliwe jest więc, że pętla taka nie wykona się ani razu, jeśli warunek od samego początku nie jest spełniony (tj. jest równy 0).

- W pętli **`do - while`** wyrażenie graniczne sprawdzane jest na końcu, po każdym obrocie pętli. Pętlę taką zawsze wykonuje się co najmniej raz.

- Najpierw wykonywane są instrukcje powtarzane, a następnie obliczane jest wyrażenie graniczne. Jeśli jest prawdziwe instrukcje wykonywane są ponownie, jeśli fałszywe pętla zostanie zatrzymana.

- Wykona się przynajmniej raz, nawet, jeśli warunek od początku jest równy 0 (tj. nie jest spełniony, np. while(n < 10) gdy n jest równe np. 300

Pętla do-while

```
long   ab = 3,  cd = 2;
//warunek przy wejściu do pętli nie jest sprawdzany, więc wykona się
// przynajmniej raz
do  {
      ab *= ab;
      cd += cd;
}
while  (ab < cd);
// wynik po zakończeniu pętli (jeden przebieg): ab == 9  cd == 4
```

Instrukcje pętli – zapętlenia

- Przykłady zapętlenia:

```
int s = 0, i; // zmienna i nie została zainicjowana
for ( int n = 0; n < 10; ++i ) // pętla nieskończona, n się nie zmienia
          // wartość i jest przypadkowa a wynik sumowania nieokreślony
    s += i;

float  A = 3.485e2,  eps = 1.38534e-2;
long  k;
while  (A != 0)
{
          A -= eps;
          ++k;
}         // ? - Pętla nieskończona, A nigdy nie będzie równe 0  ( !!! )


unsigned char  k = 5;
do
   k -= 2;
while (k != 0);       // ? - Pętla nieskończona, k nigdy nie będzie równe 0
```

Przykłady zastosowania pętli

- Pętle for, while, do – while

Instrukcje pętli – Przykład 1

- **Przykład – konwersja tekstu do liczby całkowitej dziesiętnej ze znakiem**

```
char* Text = "   -1574 "; // konwertujemy na liczbę
int X = 0;
bool sign = false,  // +, czyli liczba dodatnia
           flag = true;

// wszystkie znaki nie będące cyfrą, + czy – są ignorowane
while (*Text && flag)
     if (*Text == '+' || *Text == '-' || *Text >= '0' && *Text <= '9')
          flag = false;
     else // przesuwamy się po tablicy znakowej za pomocą wskaźnika
          Text++;

if (flag) { // flaga ma wartość true, więc doszliśmy do końca łańcucha
     printf("\nTo nie jest liczba.\n");
     return;
}
```

Instrukcje pętli – Przykład 1 cd.

```
if (*Text < '0') {          // kod ASCII znaku + lub -
      if (*Text == '-')
           sign = true;               // -, liczba ujemna
      Text++;
}
// za liczbą w łańcuchu mogą być inne znaki, np. spacje
while (*Text >= '0' && *Text <= '9')
      X = X * 10 + *Text++ - 0x30; // X = X * 10 + *Text++ - '0'
      // 0x30 to szesnastkowy kod znaku 0;
      // *Text++ wskazuje kolejne cyfry i przesuwa się 1 znak dalej
      // (*Text++ - 0x30) odejmuje od kolejnych cyfr w tablicy/łańcuchu
      // Text ich kod ASCII co daje w wyniku odpowiadające im cyfry
      // dziesiętnie, np. znak '1' zamienia w cyfrę 1.
      // Kolejne iteracje, wartość X: 1; 1*10 + 5; 15*10 + 7; 157*10 + 4
      // Wynik: X = 1574

if (sign)
      X = -X; // Wynik: X = -1574

printf("\nX = %d\n\n", X);
```

```
X = 1
X = 15
X = 157
X = 1574

X = -1574
```

Instrukcje pętli – Przykład 2

- **Przykład – konwersja dla liczb całkowitych dziesiętnych ze znakiem do tekstu**

```
int X = -31594;              // wartość przykładowa
                             // int X = INT_MIN;
                             // minimalna wartość int z biblioteki limits.h
int Weight = 1000000000;     // 10E9 - początkowy dzielnik,
                             // dla int wartość maksymalna, ok. 2.4 miliarda

printf("\nX = ");            // początek wyniku

if (X == INT_MIN){           // dla przypadku INT_MIN
      printf("-2147483648\n\n"); // wynik od razu znany
      return;                // koniec programu
}

if (X < 0){                  // dla wartości ujemnej
      _putch('-');           // wyświetlenie znaku minus i obliczenie wartości przeciwnej
      X = - X;               // nie działa dla INT_MIN, ponieważ nie istnieje liczba int
                             // o wartości 2147483648 (INT_MAX to liczba 2147483647)
}
```

Instrukcje pętli – Przykład 2 cd.

```
                // poszukiwanie liczby cyfr znaczących liczby i dostosowanie dzielnika
if (X < Weight){          // gdy X >= 1E9 dzielnik się nie zmieni
                          // dopóki zdziesiątkowany dzielnik jest jeszcze zbyt duży
      while (Weight / 10 > X) // np. dla 31594 właściwy dzielnik to 10000
            Weight /= 10;  // gdy tak zdziesiątkuj go
      Weight /= 10;       // końcowe zdziesiątkowanie
}

if (Weight == 0)          // do poprawnej obsługi wartości 0, żeby nie dzielić przez 0
      _putch('0');
else
      while (Weight >= 1) { // obliczenie i wyprowadzenie kolejnych cyfr
                            // wynikiem dzielenia całkowitego jest całkowity 0 - 9
                            // dodanie 0x30 tworzy kod ASCII cyfr 0 – 9
                            // kolejne wypisywane znaki: 31594/10000=3, 1594/1000=1,
                            // 594/100=5, 94/10=9, 4/1=4 czyli: 3 1 5 9 4
            _putch(X / Weight + 0x30);
            X %= Weight;    // pozostała część liczby, bez pierwszej cyfry
            Weight /= 10;   // zdziesiątkowanie dzielnika
      }
_putch('\n\n');
```

Instrukcje pętli – Przykład 3

- **Przykład – konwersja tekstu do liczb heksadecymalnych bez znaku**

```
char* Text = "   A1b2C3  "; // konwertujemy na liczbę
      unsigned int X = 0;
      bool flag = true;

      // wszystkie znaki nie będące cyfrą lub dużą literą są ignorowane
      while (*Text && flag)
           if (*Text >= '0' && *Text <= '9' ||
                     (*Text & 0x5F) >= 'A' && (*Text & 0x5F) <= 'F')
                flag = false;
                else // przesuwamy się po tablicy znakowej za pomocą wskaźnika
                     Text++;

                    // flaga ma wartość true, więc doszliśmy do końca łańcucha
      if (flag) {
           printf("To nie jest liczba.");
           return;
      }
```

Instrukcje pętli – Przykład 3 cd.

```
// wykorzystanie operatorów i masek bitowych
// za liczbą w łańcuchu mogą być inne znaki, np. spacje
while (*Text >= '0' && *Text <= '9' || (*Text & 0x5F) >= 'A' && (*Text & 0x5F) <= 'F') {
      // Ogólnie idea: X = X * 16 + (cyfra lub litera);
      // X <<= 4 równoważne X = X * 16;
      X <<= 4;                          // np. A * 10₁₆ (16₁₀) = A0; A1 * 10₁₆ (16₁₀) = A10
      if (*Text <= '9')                 // cyfra 0-9
            X |= *Text++ - 0x30;        // A0+1=A1
      else                              // litera A-F
            X |= (*Text++ & 0x5F) - 0x30 - 7;   // X=A10+B=A1B
                                                //* X = X | ((*Text++ & 0x5F) - 0x30 – 7)
                                                // Np. A10:101000010000
                                                // B-0x30–7 (równe 11₁₀):000000001011
                                                // wynik:101000011011 czyli A1B*/
}
// Kolejne iteracje, wartość X: A; A0 + 1; A10 + B;
// A1B0 + 2; A1B20 + C; A1B2C0 + 3
// Wynik: A1B2C3
printf("\nX = %X\n", X);
```

```
X = A
X = A1
X = A1B
X = A1B2
X = A1B2C
X = A1B2C3

X = A1B2C3
```

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Instrukcje pętli – Przykład 4

- **Przykład – konwersja liczb heksadecymalnych bez znaku do tekstu**

```
int X = 0x9A0B7C;          // wartość przykładowa
                           // maska początkowa 4 jedynki na końcu słowa n-bitowego
unsigned int Mask = 0xF;   //00000000000000000000000000001111
int L = sizeof (int) * 8; // # of bits - zazwyczaj dla int równe 32
unsigned char Char;        // kod ASCII kolejnej cyfry
any = false;               // flaga wskazująca, czy była już cyfra różna od 0
                           // (licząc od najstarszej cyfry)

printf("\nX = ");          // początek wyniku

                // przesunięcie 4 jedynek maski na początek słowa n-bitowego
                // czyli na pozycję najstarszej cyfry hex
                //11110000000000000000000000000000
Mask = Mask << (L - 4); // dla L = 32 o 28
```

Instrukcje pętli – Przykład 4 cd.

```
//L/4=32/4=8 - liczba cyfr hex w słowie n-bitowym
for (int i = 0; i < L >> 2; i++){ //8 cyfr hex maksymalnie
      // wycięcie 4 bitów kolejnej cyfry hex z X i przesunięcie ich w
      // prawo na najmłodszą pozycję słowa n-bitowego
      // dla L = 32 oraz i = 0 przesunięcie o 32 - 4 czyli o 28
      // dla i = 1 przesunięcie o 32 - 8 czyli o 24,
      // dla i = 2 przesunięcie o 32 - 12 czyli o 20,    itd.
      // wyzeruje wszystkie bity poza tymi z maski (w pierwszej iteracji nie
      // wyzeruje tylko 4 najstarszych bitów czyli bitów odpowiadających
      // pierwszej cyfrze hex lub 0)

      /* Np. X =9A0B7C:   00000000100110100000101101111100
                    Mask: 00000000111100000000000000000000
       wynik dla Mask&X:  00000000100100000000000000000000
                          czyli 900000

      dla i=2: (Mask&X) >> (32-(2 + 1)*4):    czyli o 20 bitów w prawo
      Wynik: 00000000000000000000000000001001 czyli Char = 9
      */
      Char = (Mask & X) >> (L - (i + 1) * 4);
```

Instrukcje pętli – Przykład 4 cd.

```
        // sprawdzenie czy kolejna cyfra jest różna od zera lub czy była
        // już starsza cyfra różna od zera
        if ( Char || any){  // żeby wypisać tylko liczbę bez niepotrzebnych
                            // zer na początku
             any = true;    // jest cyfra różna od zera
             if (Char > 9)  // aby uzyskać kod ASCII dla cyfr hex A – F należy
                            // oddać o 7 więcej niż dla cyfr 0 - 9
                            // trzeba przeskoczyć kody innych znaków
                            // znajdujących się pomiędzy 9 a A
                            // w kodach ASCII
                  _putch(Char + 0x37); // dla A uzyskamy 10₁₀ + 0x37 (55₁₀) = 65₁₀
             else
                  _putch(Char + 0x30); // dla 9 uzyskamy 9₁₀ + 0x30 (48₁₀) = 57₁₀
        }
        // np. na 00000000000011110000000000000000
        Mask >>= 4; // 4 jedynki maski w prawo na pozycję następnej cyfry
    }
    if (!any) putch('0'); // nie było cyfr różnych od zera
    _putch('\n');
```

Instrukcje pętli – Przykład 5

- **Przykład – konwersja liczb dziesiętnych do binarnych i odwrotnie**

  Opracować program, który wczytuje liczby całkowite dodatnie zapisane jako dziesiętne/binarne i wyprowadza ich wartości w postaci binarnej/dziesiętnej. Liczby dziesiętne poprzedzone są literą D (np. D35409), liczby binarne poprzedzone są literą B (np. B110100010).

Instrukcje pętli – Przykład 5

```
#include <conio.h>

int main(){
      unsigned int liczba = 0;
      //2^31 czyli 10000000000000000000000000000000
      unsigned int maska = 0x80000000; //ustawia najstarszy bit na 1
      unsigned long long li;

      int dalej = 1, jest = 0;
      char znak = 'X';
      int licz;

      printf("Wprowadz liczbe binarna Bxxxx lub dziesietna Dxxxxxx :\n");

      //tak długo, aż użytkownik nie poda znaku B,b lub D,d
      while(znak != 'B' && znak != 'D')       {
            znak = getche(); //pobierz znak i go wyświetl
            //zamień małe litery na duże i podstaw wynik pod znak
            znak &= 0x5F;
      }
```

Instrukcje pętli – Przykład 5 cd.

```
switch (znak){
case 'B' :                  //konwertuj binarną na dziesiętną
      while(dalej) {        //aż cała liczba zostanie wczytana
            znak = getche();          // wczytuje znak po znaku
            if (znak != '0' && znak != '1'){
                        //wypisz całą liczbę jako dziesiętną
                  printf("\nD %d\n", liczba);
                  dalej = 0;
            } else {
                        //przesuwamy bity w lewo o 1 (mnożenie razy 2)
                  liczba <<= 1;
                        // (znak - 0x30) zamiana kodu ASCII cyfry na samą
                        // cyfrę
                  liczba |= znak - 0x30;
            }
      }
      break;
      // dla wartości 1001, kolejno: 0+'1'-'0'=1, 10+0=10, 100+0=100,
      // 1000+1=1001 -> wynik 9
```

```
1       liczba = 1
0       liczba = 2
1       liczba = 5
1       liczba = 11
1       liczba = 23

bin =  10111    liczba =  23
```

Instrukcje pętli – Przykład 5 cd.

```
    case 'D' : // konwertuj dziesiętną na binarną
          scanf("%d", &liczba);
          licz = 32; // typ int jest 32 bitowy
          printf("\nB ");
          while(licz != 0)    { // bit po bicie odczytuje
               if ((liczba & maska) == 0){
                              // wypisujemy 0 dopiero po pojawieniu się
                              // pierwszej jedynki (przejrzystość)
                    if (jest) putch('0');
               } else {
                    putch('1');
                    jest = 1;
               } //dzielenie przez 2, przesunięcie o 1 bit w prawo
               maska >>= 1;
               -- licz;
          }
          putch('\n');
          break;
    }
    printf("\n\n");
    return 0;
}
```

Przerywanie pętli: break, continue

- Teoria, przykłady

Instrukcja **break**

- Czasami przydaje się możliwość wyjścia z pętli (**for**, **do**, **while**) lub z instrukcji wyboru **switch**, niezależnie, czy warunek pętli jest spełniony czy nie.

- Instrukcja **break** powoduje natychmiastowe opuszczenie ***NAJBARDZIEJ ZAGNIEŻDŻONEJ*** pętli (**for**, **do**, **while**) lub instrukcji **switch**, w której występuje. Np.:

```
for (int i = 0; i < n; i++) { // ...
     // wyjście z pętli for, nie ważne jaką wartość ma i
     if (inna_zmienna == 10)
          break;
}
```

- Przykład: funkcja *trim* usuwająca znaki odstępu, tabulacji i końca linii występujące na końcu tekstu. Instrukcja **break** służy do opuszczenia pętli po wykryciu pierwszego od końca znaku różnego od takiego znaku (\t, \n,):

```
/* usuń znaki spacji, tabulacji i nowego wiersza */
int trim(char s[]){
      int n;
      for (n = strlen(s) - 1; n >= 0; n--) {
           if (s[n] != ' ' && s[n] != '\t' && s[n] != '\n')
                break;
      }
      s[n+1] = '\0';
      return n;
}
```

Instrukcja **continue**

- W ogólności jej wywołanie powoduje natychmiastowe przerwanie danej iteracji pętli (**for**, **do**, **while**) i rozpoczęcie następnej iteracji wewnątrz tej samej pętli.
- W przypadku pętli **do** oraz **while** powoduje ona natychmiastowe sprawdzenie warunku zakończenia pętli (i jeśli wyrażenie sterujące jest wciąż różne od zera, pętla zaczyna kolejną iterację od nowa).
- W przypadku pętli **for** powoduje ona natychmiastowe przejście do trzeciej sekcji instrukcji **for** (wyrażenia zliczającego).
- Przykład: wyświetla tylko kwadraty z liczby *i* jeżeli są one nieparzyste:

```
for( i = 0; i < 100; i++ ){
      int x = i * i;
      if(x % 2 == 0)
                continue;
      printf("%d", x); // wykona się tylko wtedy, gdy x jest nieparzyste
}
```

Kompilacja, prekompilacja

- Oraz trochę o funkcji **main (...)**

Preprocesor (prekompilator)

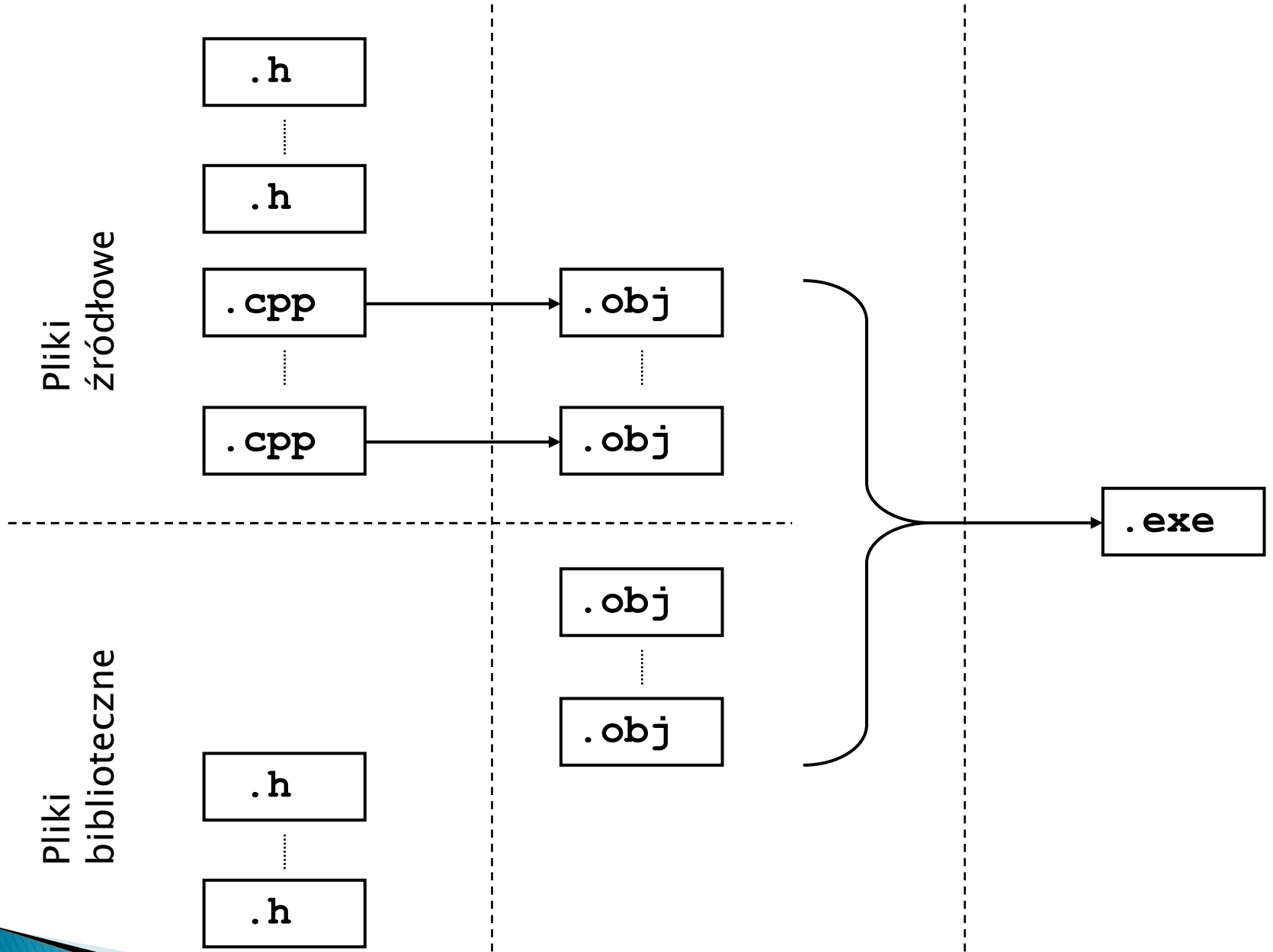
- Preprocesor zajmuje się programem zanim rozpocznie się kompilacja.

- W oparciu o **dyrektywy** (polecenia skierowane do preprocesora) zamienia on obecne w programie symboliczne skróty na ich definicje (rozwinięcia). Nie są to instrukcje, więc nie kończą się średnikiem.
- Potrafi dołączać wybrane pliki oraz określać, które części kodu są widoczne dla kompilatora.

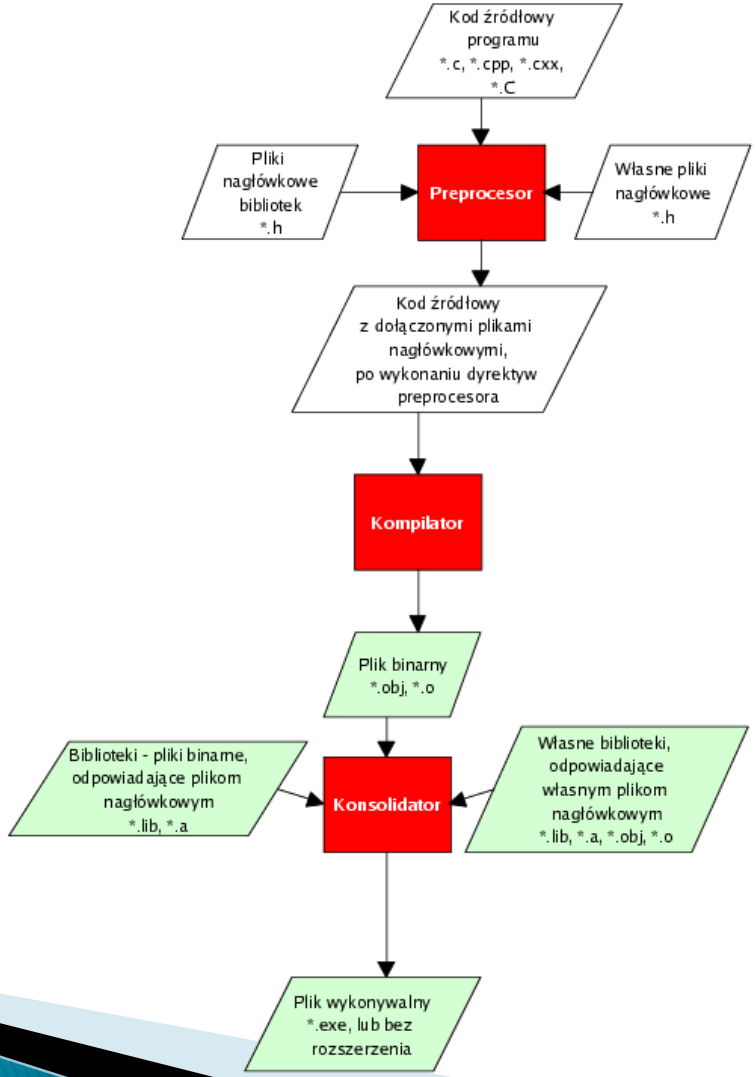
- Wiersze rozpoczynające się znakiem **#** służą do komunikacji z preprocesorem. Ich składnia jest niezależna od reszty języka i mogą wystąpić w dowolnym miejscu w programie.

- Składnia: **# *dyrektywa argumenty***
- Dwa znane już polecenia preprocesora to **#define** oraz **#include**.

- **Dołączanie plików bibliotecznych**, dyrektywa: **`#include`** Oznacza, że w miejscu wystąpienia polecenia *#include <nazwa_pliku>* zostanie wstawiona zawartość pliku określonego przez *nazwa_pliku*.
  - **`#include <stdio.h>`** – plik będzie poszukiwany zgodnie z zasadami obowiązujacymi w implementacji
  - **`#include "funkcje.h"`** – poszukiwanie pliku zaczyna się tam, gdzie znaleziono program źródłowy, a jeżeli w tym miejscu go nie ma, to plik jest poszukiwany zgodnie z zasadami obowiązującymi w implementacji.


.h
.h
.cpp
.cpp
Pliki źródłowe
.obj
.obj
.exe
.obj
.obj
Pliki biblioteczne
.h
.h
Pliki tekstowe
Pliki półskompilowane
Plik wykonywalny

Proces tłumaczenia kodu źródłowego na kod maszynowy



Preprocesor – zastępowanie tekstów: #define

- Definicja o następującej postaci:

```
#define  nazwa zastepujacy-tekst
```

spowoduje, że każde dalsze wystąpienie *nazwa* będzie zastępowane przez ciąg znaków tworzący *zastepujacy-tekst*.

- **Przykłady:**

```
#define  ROZMIAR    150
#define  moje cena_zakupu – cena_sprzedazy
#define  EPS    3.5E-8
.........
#undef   EPS                  // usunięcie definicji
#define  EPS    1.5E-8
```

- Można w ten sposób definiować dowolne nazwy i zastępować je dowolnym tekstem, np.:

```
#define forever for( ; ; ) // pętla nieskończona
```

w tym momencie każde dalsze wywołanie 'forever' w programie zostanie zastąpione przez preprocesor formułą 'for (; ;)' która jest pętlą nieskończoną.

Makrogeneracja (makra, makrorozwinięcia)

- Możliwe jest definiowanie makr z argumentami, czyli zastępujący tekst może być różny dla różnych makrowywołań. Np.:

```
#define max(A, B)  ((A) > (B) ? (A) : (B))
```

  - Wygląda jak wywołanie funkcji, ale nim nie jest.
  - Każde odwołanie (wywołanie) do *max* wymagać będzie podania dwóch argumentów (wyrażeń) i spowoduje wstawienie rozwiniętego tekstu makra bezpośrednio do tekstu programu.
  - Każde wystąpienie parametru *(A* lub *B)* zostanie zastąpione przez aktualny argument.
  - Np., wywołanie:

```
x = max(p+q, r+s);
```

  - zostanie zastąpione przez:

```
x = ((p+q) > (r+s) ? (p+q) : (r+s));
```

- Nie ma potrzeby podawania różnych definicji *max* dla różnych typów danych (jak np. dla funkcji), pod warunkiem, że traktujemy argumenty w sposób konsekwentny.

- Wyrażenia mogą być obliczane dwukrotnie lub źle jeśli powodują efekty uboczne.

Makrogeneracja (makra, makro-rozwinięcia)

- Np. wywołanie:

```
max(i++, j++)
```

Zostanie zamieniony na:

```
((i++) > (j++) ? (i++) : (j++));
```

wartość większa zostanie zwiększona dwukrotnie - błąd.

- **Przykład:**

```
#define Makro1(x) x = sin(x) + 3 * x;
.........................
double akr = 2.544;
Makro1(akr)                          // akr = sin(akr) + 3 * akr;

#define Makro2(x, y) x = x + y - 1;
.........................
double alfa = -12.74, beta = 0.21;
Makro2(alfa, beta)                   // alfa = alfa + beta - 1
 .........................
#define square(x) x * x;    // BŁĄD. Przy wywołaniu square(z+1): z+1*z+1
square(z+1);                // gdyby w definicji makra były nawiasy byłoby ok
```

Kompilacja warunkowa

- W fazie preprocesora można sterować procesem tłumaczenia przy pomocy instrukcji warunkowych. Pewne fragmenty kodu są włączane do programu zależnie od wartości warunków obliczanych podczas kompilacji.

- Polecenia preprocesora:

  - **#if** – obliczana jest wartość stałego wyrażenia całkowitego
  - **#endif** - oznacza to samo, co w normalnej instrukcji *if* nawias zamykający blok: *}*
  - **#elif** - *else if*
  - **#else**
  - **defined**(nazwa) – daje wartość prawdy (1) jeżeli *nazwa* została już wcześniej zdefiniowana (poprzez **#define**), 0 w przeciwnym wypadku.
  - **#ifndef**, **#ifdef** – sprawdzenie czy nazwa została już uprzednio zdefiniowana.

Kompilacja warunkowa

- Ogólna postać:

```
#if    wyrażenie_stałe_1
         tekst_źródłowy_1
#elif  wyrażenie_stałe_2
         tekst_ źródłowy_2
.........................
#else
       tekst_ źródłowy_n
#endif
```

- **Przykład** – zapewnienie, że zawartość pliku nagłówkowego jest wstawiana do programu źródłowego tylko raz:

```
#if ! defined(HDR)
#define HDR
/* zawartość pliku hdr.h jest tutaj */
#endif
/* oznacza warunek: jeżeli (#if) nieprawdą jest (!) że nazwa HDR została już ustalona ( defined(...)
), wtedy zdefiniuj HDR ( #define(HDR) ) */
```

Test zdefiniowania

- **Sprawdzenie czy nazwa została już uprzednio zdefiniowana**

```
#defined  identyfikator
// 1 : gdy identyfikator był już zdefiniowany
// 0 : gdy identyfikator jest nie zdefiniowany

#if   defined    identyfikator
/* równoważne */
#ifdef   identyfikator

#if   ! defined   identyfikator
/* równoważne */
#ifndef   identyfikator
```

- **Przykład:**

```
#define  WersjaProbna
//
#ifdef    WersjaProbna
. . . . . .
#else
. . . . . .
#endif
```

Funkcja main – jaka jest różnica?

- W standardzie **C99** dozwolone są tylko:

```
int main ( void )
// prawidłowe poinformowanie kompilatora, że funkcja nie
// pobiera żadnego argumentu z linii komend
int main ( int argc, char *argv[] )
```

- W standardzie **C89** użycie **`int main ( )`** jest dopuszczalne, ale zaleca się stosowanie standardu **C99**

- Funkcja *main* powinna zwracać wartość.

- Niepoprawne z punktu widzenia standardu i niezalecane, ale akceptowane przez większość kompilatorów (czynią program mniej przenośnym – może nie działać na innych systemach):

```
void main( void )
{ ... }
      // Bjarne Stroustrup:
      // " It is not and never has been C++, nor has it even been C."
void main ( )
{ ... }
```

Funkcja main

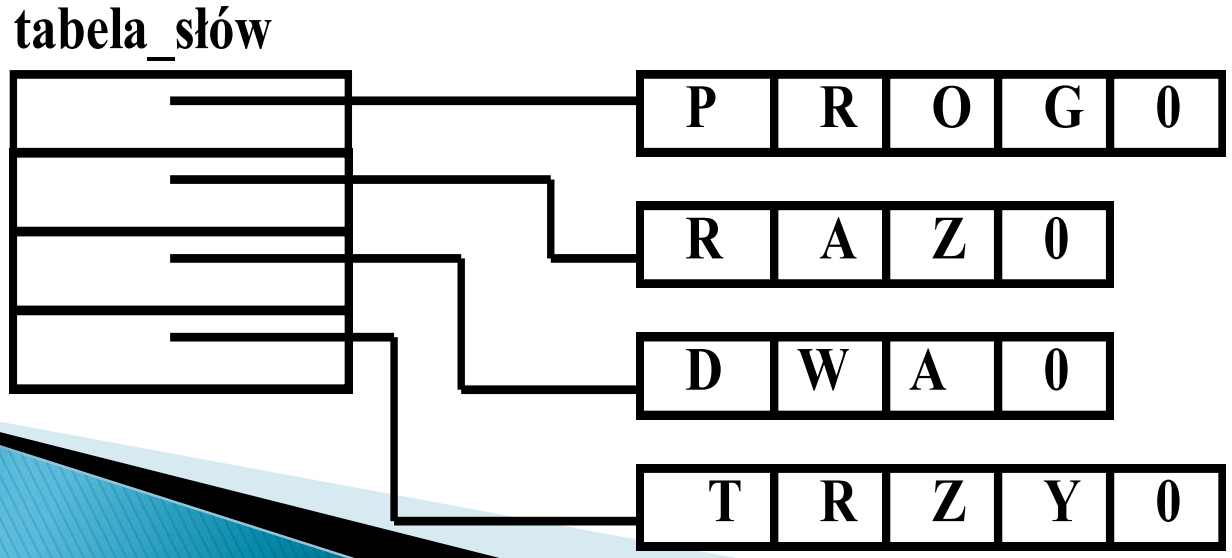
- Przykładowa definicja funkcji *main*:

```
int main ( int liczba_slow, char *tabela_slow [  ] )
{ ... }
```

  - **Wywołanie:**

PROG RAZ DWA TRZY

liczba_słów : 4



Funkcja main

- **Przykład:**

```
int main (int LiPa, char* TaPa[]){
      int index;
      if ( LiPa < 2 )
      {
            printf("\n Brak parametrów.\n\n");
            return;
      }
      for ( index = 1; index < LiPa; index++)
            printf ("\n Parametr nr %d : %s", index, TaPa[index]);

      printf("\n\n");

}
```

Pytania?