Instrukcje wejścia i wyjścia
W C i C++:
instrukcje wejścia - funkcje: **getchar(), gets(), scanf()**
Instrukcje wyjścia – funkcje: **puchar(), puts(), printf()**
Wejście:
getchar() – wprowadzenie pojedynczego znaku z klawiatury do pamięci komputera: c=getchar();
Można korzystać a typu char lub int
char znak; znak=getchar(); int litera; litera = getchar(); putchar(znak); puchar(litera);
gets() – wprowadzenie łańcucha do zmiennej - jednej linii tekstu
char nazwa[25]; gets(nazwa); puts(nazwa);
scanf() – uniwersalny sposób wprowadzania do komputera informacji wszelkiego typu.
Funkcja skanuje klawiaturę, badając czy został naciśnięty jakiś klawisz, potem interpretuje dane wejściowe zgodnie ze specyfikatorami formtowania
scanf("łańcuch formatujacy", lista danych);
Specyfikatory formatowania: **%d** – wprowadza liczbę całkowita, **%u** – liczbę bez znaku, **%f** – liczbę rzeczywistą typu float, **%e** – liczbę w zapisie wykładniczym, **%g** – liczbę dziesiętną w najkrótszym zapisie, **%c** – znak, **%s** – łańcuch, **%o** – liczba ósemkowa, **%x** – liczba szesnastkowa
int ilosc; puts("Podaj ilość "); scanf(„%d", &ilość);
char litera; scanf("%c", &litera);
char nazwa[25]; puts("Podaj nazwisko: "; scanf("%s", nazwisko);
Wyjście:
Funkcja **putchar()** wyświetla na ekranie wartość pojedynczego znaku: literał znakowy, stałą typu char, zmienną typu char.
puchar('H');
#define INICJAL 'H';
main() { putchar(INICHAL); }
main() { char litera; litera='G'; putchar(litera); }
Funkcja **puts()** – wyświetla na ekranie łańcuch znaków: literał łańcuchowy, stałą łańcuchową, zmienną łańcuchową.
Literał znakowy: puts(„Jak się masz");
Stała łańcuchowa:
#define KOMUNIKAT "Jak się masz"
main() { puts(KOMUNIKAT); }
Zmienna łańcuchowa:
char pozdrowienie[] „Jak się masz"
main() { puts(pozdrowienie) }
Uniwersalna funkcja **printf()** – może wyświetlać dane dowolnego typu i współpracować z wieloma argumentami
printf(„łańcuch formatu", lista argumentów);
Łańcuch formatu, inaczej łańcuch sterujący, w cudzysłowie, zawiera dowolny tekst wraz ze specyfikatorami formatowania, wskazującymi typ i położenie danych. Specyfikatory formatowania takie jak dla instrukcji scanf().
char uczen[] = „Nowak Jan"; wiek=17; printf("Uczen %s ma %d lat ", uczen, wiek);

Operatory
Operatory arytmetyczne: * / + - % (reszta z dzielenia liczb całkowitych)
Operatory relacyjne: < <= > >= == !=
Bitowe operatory logiczne
Bitowe operatory logiczne działają na uporządkowanym ciągu bitów, które przyjmują wartości 1 lub 0.
& - bitowa koniunkcja
! – bitowa alternatywa
^ - bitowa różnica symetryczna
~ - bitowa negacja
<< - przesuniecie w lewo
>> - przesuniecie w prawo
Operator przesunięcia w lewo <<
Operator przesunięcia w prawo >>
Operator << przesuwa wszystkie bity argumentu w lewo, natomiast >> przesuwa je w prawo.
Przesunięciu liczby w lewo o jeden bit odpowiada pomnożenie jej przez dwa, natomiast przesunięcie w prawo podzieleniu jej przez dwa (całkowicie).
Przykłady:
a = a << 1; // pomnożenie zmiennej a przez 2^1, czyli 2
```
a = a << 2; // pomnożenie zmiennej a przez 2^2, czyli 4
a = a << 3; // pomnożenie zmiennej a przez 2^3, czyli 8
a = a >> 1; // podzielenie zmiennej a przez 2^1, czyli 2
a = a >> 2; // podzielenie zmiennej a przez 2^2, czyli 4
```
Przesunięcia bitowe są o wiele szybsze, dzięki czemu można zwiększyć szybkość działania swojego programu.

Operatory zwiększania ++ i zmniejszania –
Np.
x = ++a; // najpierw zwiększamy wartość zmiennej a o 1 a następnie przypisujemy zwiększoną wartość zmiennej z
x = a++; // najpierw dokonujemy przypisania a dopiero później zwiększamy wartość zmiennej a.
Reguła ta dotyczy też operatora –
Przykład:
int a, x; x=100; a=10; x=a++ // x→10, a→11
x=++a; // a→ 12, x→ 12
Operator ++ wykorzystywany w postaci x = ++a; nazywa się przedrostkowym a x = a++; przyrostkowym
a=10; a++ // a→ 11
a=10; ++a // a→11

Wieloznakowe operatory przypisania typu: **#=**, gdzie # oznacza **+, -, *, /, %, <<, >>, &, !, ~**
Czyli np. postaci a #= b oznacza a = a#b;
a += b; // a = a+b
a -= b; // a = a-b
a *= b; // a = a*b
a /= b; // a = a/b

Operator przecinkowy
Służy do utworzenia wyrażenia złożonego z ciagu wyrażeń składowych oddzielonych przecinkami.
Kolejno oblicza się wartości wyrażeń składowych o strony lewej do prawej.
Wyrażenie złożone użyte w instrukcji przypisania musi być ujęte w nawiasy.
Przykład:
int a, b;
a=10; b = (a++, a+=10); *// a++ →11; a = a + 10 →21 - zostanie przypisana b wartość 21*

Hierarchia i łączność operatorów
Operatory posiadają określone priorytety. Część priorytetów jest taka sama jak w matematyce. W przypadku wątpliwośći można użyć nawiasów lub sprawdzić priorytet w systemie.
Pewne operatory wiążą od strony lewej do prawej a inne od strony prawej do lewej.
Np. operator przypisania wiąże od strony prawej do lewej, a **operator +** od strony lewej do prawej.
Przykład:
int a, b, w;
b=1; a=10; w=a=b; // zmiennej w zostanie przypisana wartość 1: a=1; w=1;
/* gdyż operator przypisania wiąże od strony prawej do lewej; najpierw przypisane a=b, potem w=a */

Operator "?"
Ogólna konstrukcja tego operatora wygląda tak:
```
(wyrażenie_logiczne) ? wyrażenie_gdy_prawda : wyrażenie_gdy_fałsz
```

Najpierw sprawdzane jest wyrażenie logiczne w nawiasie,
jeśli jest ono prawdziwe, to obliczana jest wartość wyrażenia znajdującego się po znaku ? i jest ona zwracana,
natomiast w przypadku, gdy wyrażenie logiczne było fałszywe to obliczana jest wartość wyrażenia znajdującego się po znaku : i właśnie ona jest zwracana.

Załóżmy, że mamy do czynienia z dwoma zmiennymi *a* i *b* i chcielibyśmy wyświetlić na ekranie wartość większej z nich.
Program przy użyciu znanej już instrukcji if wyglądałby tak:
```
#include <stdio.h>
void main(void)
{
  int a = 5, b = 6, c;
  if(a > b) c=a; else c=b;
  printf("Max(a,b) = %d\n", c);
}
```
Program jest prosty - gdy zmienna *a* jest większa od zmiennej *b* to przypisuje zmiennej *c* wartość zmiennej *a*, natomiast w przeciwnym wypadku przypisuje jej wartość zmiennej *b*.
Następnie na ekranie wyświetla wartość zmiennej *c*.

Zapis można uprościć stosując operator ?
```
#include <stdio.h>
void main(void)
{
  int a = 5, b = 6, c;
  c = (a > b) ? a : b;
  printf("Max(a,b) = %d\n", c);
}
```

W naszym programie najpierw sprawdzamy, czy zmienna *a* jest większa od zmiennej *b*.
Ponieważ *a* jest równe pięć, natomiast *b* jest równe sześć to wyrażenie to jest fałszywe i obliczana jest wartość wyrażenia występującego po znaku :
W naszym przypadku jest tam po prostu zmienna *b*, której wartość jest zwracana i przypisana do zmiennej *c*.
Program można zapisać także w bardziej skrócony sposób:

```
#include <stdio.h>
void main(void)
{
  int a = 5, b = 6;
  printf("Max(a,b) = %d\n", (a>b) ? a : b);
}
```

Wartość wyrażenia jest od razu przekazywana do funkcji **printf**.
Czegoś takiego przy użyciu instrukcji if nie da się zrobić, bo nie zwraca ona żadnej wartości, a jedynie, w zależności od warunku, wykonuje te, lub inne instrukcje.

Rzutowanie

Czasami mając zadeklarowaną zmienną jednego typu, musimy jej użyć jako inny typ.
Przykładowo stworzyliśmy zmienną typu **int** a mamy funkcję która obsługuje tylko typ **float.**
Co wtedy? Trzeba podać wartość zmiennej int jako wartość float.
Na tym polega rzutowanie w C++**.**

Rzutowanie niejawne

Tworząc bądź modyfikując zmienną np. typu float możemy jej przypisać wartość typu int.

```
float liczba = 12;
//lub
liczba = 12;
```

Tą zmienną powinniśmy zadeklarować jako 12.0, ale użyliśmy (niejawnie) rzutowania i błędu nie ma.

Rzutowanie w stylu C

Rzutowanie niejawne może nie zawsze działać.
Po za tym jest niewygodne, gdyż nie widzimy, kiedy go używamy. Za to można rzutować zmienną na określony typ.

Przykład:

```
float liczba = 13.4;
int liczba_int =( int ) liczba;
```

Inna forma

```
float liczba = 13.4;
int liczba_int = int( liczba );
```

Problem – dzielenie liczb całkowitych, wynik jest niewłaściwy (zaokrąglony do liczb całkowitych), mimo deklaracji zmiennej wynikowej typu float

```
void main(void)
{
  int a = 5, b = 3;   float r;
  r = a / b;
  printf("r = %f\n", r);
}
```

Wynik tej operacji to nie było to "1.666667", lecz **"1.000000".**
Działo się tak, dlatego, że argumenty operacji dzielenia były typu całkowitego, a ten, nie przechowuje informacji o części ułamkowej. W efekcie wynik operacji także był całkowity i ta właśnie całkowita wartość była przypisana zmiennej *r***.**
Aby temu zaradzić można wykorzystać tzw. rzutowanie.
Jest to tymczasowa, tylko dla potrzeb obliczenia danego wyrażenia, zmiana typu.
Typ ten jednak nie zmienia się fizycznie, po prostu kompilator traktuje zmienną danego typu tak, jakby była typu, na który ją rzutujemy**.**
Program działający prawidłowo, z zastosowaniem, **rzutowania**, można zapisać następująco:

```
#include <stdio.h>
void main(void)
{
  int a = 5, b = 3;   float r;
  r = (float)a / b;
  printf("r = %f\n", r);
}
```

Przed zmienną *a* przy operacji dzielenia dodany został taki oto ciąg znaków: "(float)".
Informuje on kompilator, że zmienna (lub wyrażenie) następujące za nim ma być traktowane tak, jakby było typu float .

W naszym przypadku mówimy kompilatorowi, żeby potraktował zmienną *a* jakby była typu rzeczywistego, (czyli tak naprawdę rozszerza ją do postaci "5.0"), a następnie tą rzeczywistą liczbę podzielił przez zmienną *b*, która jest typu całkowitego.
Podsumowując, rzutowanie wykorzystujemy, aby przy operacjach przypisania do siebie różnych typów poinformować kompilator, że wiemy, co robimy i ma on traktować to zgodnie z naszym życzeniem.
Aby dokonać rzutowania danej zmiennej, czy wyrażenia na inny typ, należy przed nią (nim) wpisać w nawiasach typ, na który chcemy rzutować.

Typ wyliczeniowy

Typ wyliczeniowy nie jest typem danych w ścisłym tego słowa znaczeniu, gdyż jest to odpowiednik typu int.
Ma on jednak ciekawą cechę, a mianowicie kolejne jego elementy możemy nazwać wedle swojego uznania.
Przykładowy program:

```
#include <stdio.h>
#include <conio.h>
void main(void)
{
 enum {Pn, Wt, Sr, Czw, Pt, Sb=10, Nd=11} DzienTyg; /* dni tyg. od 0,zmiana Sb. */
  DzienTyg = Pn; printf("Wartosc dla poniedzialku = %d\n", DzienTyg);
  DzienTyg = Wt; printf("Wartosc dla wtorku = %d\n", DzienTyg);
  DzienTyg = Sr; printf("Wartosc dla srody = %d\n", DzienTyg);
  DzienTyg = Czw; printf("Wartosc dla czwartku = %d\n", DzienTyg);
  DzienTyg = Pt; printf("Wartosc dla piatku = %d\n", DzienTyg);
  DzienTyg = Sb; printf("Wartosc dla soboty = %d\n", DzienTyg);
  DzienTyg = Nd; printf("Wartosc dla niedzieli = %d\n\n", DzienTyg);
  /* miesiące */
  enum {Sty=1, Lut, Mar, Kwi, Maj, Czer, Lip, Sie, Wrz, Paz, Lis, Gru} Miesiac;
  Miesiac = Kwi; printf("Miesiac Kwiecien to  %d  mc roku \n", Miesiac);
  enum {Pon=1, Wto, Sro, Czwa, Piat, Sob, Niedz} DzTyg; /* dni tygodnia od 1 */
  DzTyg= Wto; printf("Wtorek  to  %d  dzien tygodnia \n",DzTyg);
  getch();
}
```

W programie tym chcemy operować na zmiennej, która będzie przechowywać dzień tygodnia. Moglibyśmy po prostu zadeklarować ją jako int i przyjąć założenie, że poniedziałkowi odpowiada wartość zero, wtorkowi wartość jeden itd. Jednak, gdy przy dalszej rozbudowie programu chcielibyśmy przypisać tej zmiennej wartość "Środa" to musielibyśmy sobie przypominać jaka liczba jej odpowiada. Możemy jednak ułatwić sobie to zadanie dzięki zastosowaniu enum . Każdej kolejnej wartości możemy przydzielić identyfikator, który łatwiej będzie zapamiętać. Jak widzisz w naszym programie zadeklarowaliśmy identyfikatory "Pn", "Wt", "Sr" itd. enum już sam zadba o przydzielenie im konkretnych wartości - tzn. "Pn" będzie odpowiadało wartości zero, "Wt" jeden itd. Jeśli z jakiegoś powodu chciałbyś, aby od pewnego identyfikatora nastąpił przeskok i żeby liczenie zaczynało się od innej wartości to podajesz ją po znaku równości.
W naszym przykładzie identyfikatorowi "Sb" przydzieliliśmy wartość dziesięć (pamiętaj, że kolejnym identyfikatorom będą odpowiadały zmienione już wartości - w naszym przykładzie niedzieli będzie przydzielona wartość jedenaście).
Teraz zamiast pisać:

```
DzienTyg = 2;
```

możemy po prostu napisać:

```
DzienTyg = Sr;
```

Ważne jest jednak, abyś pamiętał, że mimo faktu przypisywania zmiennej wartości poprzez nadane identyfikatory, to nadal są to zwykłe liczby.

Instrukcja typedef

Do deklaracji nowego typu danych służy instrukcja typedef .
Przykład:

```
#include &ltstdio.h>
typedef float rzeczywista;
void main(void)
{
  rzeczywista a=4.5;
  printf("%f\n", a);
}
```

Ogólna postać deklaracji **typedef**:

```
typedef definicja_typu nazwa_nowego_typu;
```

W naszym przypadku przykład deklaracji nowego typu danych, któremu nadaliśmy nazwę "rzeczywista", mamy w drugiej linijce programu.
Określiliśmy tam, że nowy typ będzie po prostu typem float tylko ze zmienioną nazwą.

Ponieważ identyfikator "rzeczywista" odpowiada od tego momentu nowemu typowi danych to możemy zadeklarować sobie zmienną tego typu.
W funkcji main deklarujemy zmienną o nazwie *a*. Ponieważ jej typ został wyprowadzony z typu float to możemy używać jej tak, jakby była to zmienna typu float. W naszym programie po prostu ją wyświetlamy na ekranie.

Pytanie - czemu służy to polecenie, skoro możemy po prostu używać wbudowanego typu float .
I. Po pierwsze, deklaracja ta może dotyczyć o wiele bardziej złożonego typu danych.
II. Po drugie, dzięki temu możemy w łatwy sposób przenieść nasz program na inny kompilator lub system.
Na przykład zmienna typu int może na jednym kompilatorze zajmować cztery bajty, a na innym tylko dwa.
Możemy w łatwy sposób temu zaradzić poprzez deklarację naszego nowego typu o nazwie na przykład "MOJINT" i wszędzie go używać. Teraz jeśli chcielibyśmy go przenieść na inny kompilator to wystarczy jedynie zmienić deklarację typu i gotowe !
III, Trzecim powodem może być potrzeba zwiększenia precyzji obliczeń.
Jeśli program wszelkie obliczenia wykonywał na zmiennych typu float to w takim przypadku musielibyśmy zmienić wszelkie wystąpienia tego typu na typ double .
A tak wystarczy jedynie zadeklarować nowy typ, na przykład "FLOAT1" i używać go zamiast float , a przy konieczności zwiększenia precyzji zmienić jedynie deklarację typu "FLOAT1" tak, żeby wyprowadzony był z typu double .

Podejmowanie decyzji w programie

Instrukcja if …else

Składnia:
if (wyrażenie) instrukcja1;
if (wyrażenie) instrukcja1; else instrukcja2;
Przykłady

```
if( a > 10 ) printf("a jest większe od dziesięciu !");
else printf(" a jest mniejsze lub równe dziesięć !");

if( a > 10 ) printf("Zmienna a jest większa od dziesięciu !");

if ( a != 0 ) printf("a jest rożne od zera");
```

możemy napisać:

```
if ( a ) printf("a jest rożne od zera");

if( a > 10 )
 {
  printf("a jest wieksze od dziesięciu !\n");
  printf("jest równe  %d.", a);
 }
else
  {
  printf("a jest mniejsze lub równe dziesięć !\n");
  if (a != 5) printf("jednak nie jest równe pięć !");
    }
```

Instrukcja switch

Jeśli zachodzi konieczność podjęcia kilku decyzji, wtedy stosujemy instrukcję switch.

Składnia instrukcji switch

```
switch(wyrażenie)
 {
   case wartosc1: instrukcje;  // wartość1 to inaczej etykieta1
   case wartosc2: instrukcje;  // wartość1 to inaczej etykieta2
   …
   default: instrukcje;
 }
```

lub

```
 switch(wyrażenie)
 {
   case wartosc1: instrukcje; break;  // wartość1 to inaczej etykieta1
   case wartosc2: instrukcje; break;  // wartość1 to inaczej etykieta2
   …
   default: instrukcje; break;
 }
```

Wykonanie instrukcji switch dla pierwszej postaci przebiega następująco: jeśli któraś z etykiet jest równa wartości wyrażenie, to sterowanie przechodzi do tej etykiety i wykonywane są wszystkie instrukcje umieszczone poniżej aż do końca

instrukcji złożonej, łącznie z instrukcjami odpowiadającymi innym etykietom. Sterowanie przechodzi do etykiety default, gdy obliczona wartość nie jest równa żadnej etykiecie.
Aby przerwać wykonanie dyrektyw umieszczonych wewnątrz instrukcji złożonej, należy wykonać instrukcję break, co przedstawia druga wersja instrukcji switch.

Załóżmy, że mamy program:

```
#include <stdio.h>
void main(void)
{
  int a = 5;
  if(a == 4) printf("Zmienna a jest rowna cztery.\n");
  else if((a == 5) || (a==6)) printf("Zmienna a jest równa piec lub szesc.\n");
  else printf("Zmienna a jest nie jest rowna ani cztery, ani piec, ani szesc.\n");
}
```

Program na podstawie wartości zmiennej *a* wypisuje na ekranie odpowiedni tekst.
Aby ten cel osiągnąć musieliśmy zastosować sekwencję instrukcji if .
W takim przypadku lepiej użyć instrukcję **switch** .

Program przy użyciu switch:

```
#include <stdio.h>
void main(void)
{
 int a = 5;
 switch(a)
 {
  case 4 : printf("Zmienna a jest rowna cztery.\n");  break;
  case 5 :
  case 6 : printf("Zmienna a jest rowna piec lub szesc.\n");  break;
  default: printf("Zmienna a jest nie jest rowna cztery, ani piec, ani szesc.\n");
  }
}
```

Organizacja obliczeń cyklicznych

Isntrukcja - pętla "while"

while (wyrażenie) instrukcje;

Najpierw sprawdzana wartość wyrażenia i jeśli jest różne od zera to wykonywana jest instrukcja
Pętla będzie wykonywana dopóty, dopóki warunek ten jest spełniony.
Instrukcja może nie być nigdy wykonana.

Przykład

```
#include <stdio.h>
void main(void)
{
  int licznik = 10;
  printf("Poczatek petli\n");
  while(licznik >= 0)
  {
    printf("Zmienna licznik = %d\n", licznik);
    licznik--;
  }
  printf("Koniec petli\n");
}
```

Instrukcja - pętla "do-while"

do instrukcja; while (wyrazenie);

Instrukcja (może być złożona) jest wykonywana tak długo, jak długo wartość wyrażenia (wyrażenie) jest różna ode zera (prawda).
Instrukcja wykona się co najmniej raz – warunek sprawdzany na końcu.
Różnicą w stosunku do instrukcji while jest to, że pętla while może się w ogóle się nie wykonać, jeśli warunek sprawdzany na początku nie był spełniony.
Przykład: program z poprzedniego punktu przy użyciu do...while:

```
/* do_while.c  */
#include <stdio.h>
```

```
void main(void)
{
  int licznik = 10;
  printf("Poczatek petli\n");
  do
  {
    printf("Zmienna licznik = %d\n", licznik);     licznik--;
  } while(licznik >= 0);

  printf("Koniec petli\n");
}
```

Różnica jest minimalna. Po while w tym przypadku stawiamy średnik.

Instrukcja - pętla "for"

Najbardziej rozbudowaną konstrukcją pętli jest pętla "for". Ogólny jej zapis wygląda następująco:

```
for (wyrażenie1; wyrazenie2; wyrazenie3) instrukcja;
```

lub inny zapis

```
for(inicjalizacja; warunek; inkrementacja) {  instrukcje do wykonania }
```

Instrukcja równoważna instrukcji:

```
wyrazenie1;
while (wyrażenie2) {  Instrukcja; Wyrazenie3; }
```

lub inny zapis

```
inicjalizacja;
while (warunek){  {instrukcje do wykonania};  inkrementacja; } }
```

Najpierw jest obliczana wartość wyrażenie1(inicjalizacja), co powoduje na ogół wyznaczenie wartości początkowej zmiennej sterującej. Następnie jest obliczana wartość wyrażenie2 (warunek) i jeśli jest różna od zera (prawda) to jest wykonywana instrukcja (może być złożona – instrukcje w nawiasach {}) oraz wyrażenie3-inkrementacja, w którym przeważnie następuje zmiana zmiennej sterującej pętli.

```
Przykład: for (int i=1; i<11; i++) instrukcja;
```

Pętla ta działa na podstawie algorytmu:

1. Wykonywane są instrukcje zawarte w części "inicjalizacja" – wyrazenie1.
Jest to wykonywane tylko jeden raz, na samym początku pętli.
2. Sprawdzany jest warunek (wyrażenie2) - jeśli jest fałszywy to następuje skok do kroku numer sześć.
3. Wykonywane są "instrukcje do wykonania" – instrukcja lub instrukcja złożona.
4. Wykonywane są instrukcje zawarte w części "inkrementacja" – wyrażenie3.
5. Następuje skok do kroku numer dwa.
6. Pętla kończy się - wykonywane są instrukcje poza pętlą.

Żaden z podanych parametrów nie jest wymagany, tak, więc najprostszą pętlą przy użyciu "for" jest:

```
for(;;) {  instrukcje do wykonania }
```

Taką konstrukcję nazywa się czasem "**forever**" ponieważ jest to pętla nieskończona (będzie wykonywała się aż do momentu, gdy użytkownik zresetuje komputer lub gdy napotka instrukcję break lub g**oto.**

Przykładowy program z poprzednich punktów przy użyciu pętli "for":

```
#include <stdio.h>
void main(void)
{
  int licznik;
  printf("Poczatek petli\n");
  for(licznik=10; licznik >= 0; licznik--)
    { printf("Zmienna licznik = %d\n", licznik);  }

  printf("Koniec petli\n");
}
```

W części inicjalizacyjnej dokonujemy ustawienia zmiennej *licznik* na dziesięć.
(Można to zrobić tak jak poprzednio - od razu ją ustawić przy deklaracji i część tą zostawić pustą).
Następnie widzimy warunek - tu nic nowego, wygląda on dokładnie tak samo jak poprzednio.
Zmniejszenia licznika nie dokonujemy jednak wewnątrz pętli, lecz w części "inkrementacja".
Oczywiście moglibyśmy to zrobić wewnątrz pętli, a tą cześć pozostawić pustą, ale zostało to wykonane w ten sposób, aby zaprezentować sposób użycia.

Części "inicjalizacja" i "inkrementacja" mogą zawierać także po kilka instrukcji - oddzielamy je wtedy przecinkami.
Przykład ze zmodyfikowaną wersję poprzedniego programu - została dodana jeszcze jedna zmienna,

która jest zwiększana z każdym przejściem pętli o dwa:

```
#include <stdio.h>
void main(void)
{
  int licznik, a;
  printf("Poczatek petli\n");
  for(licznik=10, a=5; licznik>0; licznik--, a+=2)
  {
    printf("Zmienna licznik = %d\n", licznik);
    printf("Zmienna a       = %d\n", a);
  }

  printf("Koniec petli\n");
}
```

Instrukcje break, continue i **goto**

Czasem może się zdarzyć potrzeba pominięcia jednego przebiegu pętli lub wcześniejszego jej przerwania.
Służą do tego celu dwie wspomniane instrukcje.
Instrukcja break jest wręcz konieczna w przypadku zastosowania pętli nieskończonej (pokazanej w poprzednim punkcie). Przyjrzyjmy się nieco zmodyfikowanemu programowi z poprzedniego punktu:

```
#include <stdio.h>
void main(void)
{
  int licznik;
  printf("Poczatek petli\n");
  for(licznik=10; licznik  0; licznik--)
  {
    if(licznik == 5) continue;
    if(licznik == 2) break;
    printf("Zmienna licznik = %d\n", licznik);
  }
  printf("Koniec petli\n");
}
```

W przypadku, gdy zmienna *licznik* jest równa pięć wykonywana jest instrukcja continue , a gdy jest równa dwa, wykonywana jest instrukcja break .
Do wartości zmiennej licznik równej sześć jest wyświetlana odpowiednia linia. Jednak linijka informująca, że zmienna licznik jest równa pięć w ogóle nie została wyświetlona - odpowiedzialna jest za to instrukcja continue .
Na początku nie jest ona wykonywana, jednak, gdy zmienna *licznik* dojdzie do wartości pięć, zostaje ona wykonana. Powoduje to, że pętla for pomija wszystkie następujące po continue instrukcje wewnątrz pętli (dla danego przebiegu) i przechodzi od razu do następnej iteracji.
W następnych dwóch przebiegach żaden z warunków występujących po instrukcjach if nie jest spełniony i program wyświetla informacje, że zmienna *licznik* jest równa, kolejno, cztery i trzy.
Dochodzimy teraz do momentu, gdy zmienna *licznik* osiąga wartość dwa.
W takim wypadku spełniony jest warunek drugiej z instrukcji if i wykonywana jest instrukcja break - pętla kończy swe działanie i zostaje wyświetlony komunikat o tym informujący.

Alternatywą instrukcji **break** jest instrukcja **goto .**
Mimo, że w przypadku języków strukturalnych, (jakim jest język C) jej stosowanie nie jest zalecane, to jest jednak sytuacja, gdy można ją zastosować, ponieważ nie istnieje żaden inny prosty sposób osiągnięcia celu.
Sytuacją tą jest wyjście z zagnieżdżonej pętli.
Przykład:

```
#include <stdio.h>
void main(void)
{
  int a, b;
  printf("Poczatek petli\n");
  for(a=0; a < 4; a++)
  {
    for(b=0; b < 4; b++)
    {  // if((a==2) && (b==1)) break;
      printf("a = %d, b = %d\n", a, b);
    }
  }
```

```
  printf("Koniec petli\n");
}
```

Jest tu zagnieżdżona pętla, tzn. jedna pętla jest wykonywana w drugiej.
Rozwiązaniem jest tutaj użycie instrukcji **goto**:

```
#include <stdio.h>
void main(void)
{
  int a, b;
  printf("Poczatek petli\n");

  for(a=0; a < 4; a++)
  {
    for(b=0; b < 4; b++)
    {
      if((a==2) && (b==1)) goto koniec;
      printf("a = %d, b = %d\n", a, b);
    }
  }
  koniec:
  printf("Koniec petli\n");
}
```

W programie tym instrukcja break została zastąpiona przez goto koniec . "koniec" jest to tzw. etykieta.
Zadeklarowaliśmy ją przed ostatnią instrukcją programu - jest to po prostu dowolna nazwa zakończona dwukropkiem.
W programie możemy zadeklarować dowolną liczbę etykiet (oczywiście każda musi mieć inną nazwę).
Aby przenieść wykonywanie programu do innego miejsca należy wykonać instrukcję goto podając jej nazwę etykiety, do której ma nastąpić skok.
W naszym przypadku po wykonaniu instrukcji goto koniec program wyświetli informację o końcu pętli i zakończy swe działanie.

Funkcje

W C **funkcja** (czasami nazywana podprogramem, rzadziej procedurą) to wydzielona część programu, która przetwarza argumenty i ewentualnie zwraca wartość, która następnie może być wykorzystana jako argument w innych działaniach lub funkcjach.
Funkcja może posiadać własne zmienne lokalne.
Główną motywacją tworzenia funkcji jest unikanie powtarzania kilka razy tego samego kodu.
W odróżnieniu od funkcji matematycznych, funkcje w C mogą zwracać dla tych samych argumentów różne wartości.
Innym, niemniej ważnym powodem używania funkcji jest rozbicie programu na fragmenty wg ich funkcjonalności.
Oznacza to, że jeden duży program dzieli się na mniejsze funkcje, które są "wyspecjalizowane" w wykonywaniu określonych czynności. Dzięki temu łatwiej jest zlokalizować błąd.
Ponadto takie funkcje można potem przenieść do innych programów.

Przykład funkcji

```
int iloczyn (int x, int y)
{
  int iloczyn_xy;
  iloczyn_xy = x*y;
  return iloczyn_xy;
}
```

int iloczyn (int x, int y) to nagłówek funkcji, który opisuje, jakie argumenty przyjmuje funkcja i jaką wartość zwraca (funkcja może przyjmować wiele argumentów, lecz może zwracać tylko jedną wartość).
Na początku podajemy typ zwracanej wartości - u nas int. Następnie mamy nazwę funkcji i w nawiasach listę argumentów.
Ciało funkcji (czyli wszystkie wykonywane w niej operacje) umieszczamy w nawiasach klamrowych.
Pierwszą instrukcją jest deklaracja zmiennej - jest to zmienna lokalna, czyli niewidoczna poza funkcją.
Dalej przeprowadzamy odpowiednie działania i zwracamy rezultat za pomocą instrukcji return.

Funkcję w języku C tworzy się następująco:

```
typ identyfikator (typ1 argument1, typ2 argument2, typ_n argument_n)
{
  /* instrukcje */
}
```

Istnieje możliwość utworzenia funkcji, która nie posiada żadnych argumentów.
Definiuje się ją tak samo, jak funkcję z argumentami z tą tylko różnicą, że między okrągłymi nawiasami nie znajduje się

żaden argument lub pojedyncze słówko void - w definicji funkcji nie ma to znaczenia, jednak w deklaracji puste nawiasy oznaczają, że prototyp nie informuje jakie argumenty przyjmuje funkcja, dlatego bezpieczniej jest stosować słówko void.
Funkcje definiuje się poza główną funkcją programu (main).
W języku C nie można tworzyć zagnieżdżonych funkcji (funkcji wewnątrz innych funkcji).

Procedura

Przyjęło się, że **procedura** od funkcji różni się tym, że ta pierwsza nie zwraca żadnej wartości.

Schemat procedury

```
void identyfikator (argument1, argument2, argument_n)
{
  /* instrukcje */
}
```

void (z ang. pusty, próżny) jest słowem kluczowym mającym kilka znaczeń, w tym przypadku oznacza "brak wartości".

Wywoływanie funkcji

Funkcje wywołuje się następująco:

```
identyfikator (argument1, argument2, argumentn);
```

Jeśli chcemy, aby przypisać zmiennej wartość, którą zwraca funkcja, należy napisać tak:

```
zmienna = funkcja (argument1, argument2, argumentn);
```

Przykładowo, mamy funkcję:

```
void pisz_komunikat()
{   printf("To jest komunikat\n"); }
```

Wywołanie:

```
pisz_komunikat(); /* dobrze */
```

Przykładowy program z funkcją:

```
#include <stdio.h>
 int suma (int a, int b)
 {    return a+b;  }

 int main ()
 {
   int m = suma (4, 5);
   printf ("4+5=%d\n", m);
   return 0;
 }
```

return to słowo kluczowe języka C.
W przypadku funkcji służy ono do:
przerwania funkcji (i przejścia do następnej instrukcji w funkcji wywołującej)
zwrócenia wartości.

W przypadku procedur powoduje przerwania procedury bez zwracania wartości.
Użycie tej instrukcji jest bardzo proste i wygląda tak:

```
return zwracana_wartość;
return;   // dla procedur
```

Możliwe jest użycie kilku instrukcji return w obrębie jednej funkcji.
Wielu programistów uważa jednak, że lepsze jest użycie jednej instrukcji return na końcu funkcji, gdyż ułatwia to śledzenie przebiegu programu.
W C zwykle przyjmuje się, że 0 oznacza poprawne zakończenie funkcji:

```
return 0; /* funkcja zakończona sukcesem */
a inne wartości oznaczają niepoprawne zakończenie:
return 1; /*funkcja zakończona niepowodzeniem */
```

Przykłady programów

```
/* Program Pprost1.c – funkcja zdefiniowana na początku */
#include <studio.h>
float PoleProstokata(float bok1, float bok2)
{  // w tym miejscu bok1 jest równy 5,  natomiast b jest równe 10
   float wynik;
   wynik = bok1 * bok2;
   return wynik;
}
void main(void)
{ float a, b, dlugosc;
   a = 5; b = 10;
   dlugosc = PoleProstokata(a, b);
```

```
}

/* Program Pprost2.c – funkcja zadeklarowana na początku
a zdefiniowana po funkcji main */
#include <studio.h>

float PoleProstokata(float bok1, float bok2); /* deklaracja funkcji */

void main(void)
{ float a, b, dlugosc;
  a = 5; b = 10;
  dlugosc = PoleProstokata(a, b);
}
/* definicja funkcji  */
float PoleProstokata(float bok1, float bok2)
{  // w tym miejscu bok1 jest równy 5,  natomiast b jest równe 10
  float wynik;
  wynik = bok1 * bok2;
  return wynik;
}

/* kolo1.c */
#include <stdio.h>
#define PI 3.1415

float ObliczPole(float promien);
float ObliczObwod(float promien);

void main(void)
{
  float pole, obwod;
  pole = ObliczPole(5);
  obwod = ObliczObwod(5);
}

float ObliczPole(float promien)
{ /* wzór na pole to PI*R^2 */
  return (PI * promien * promien);
}

float ObliczObwod(float promien)
{
  /* wzór na obwód to 2*PI*R */
  return (2 * PI * promien);
}
```

Funkcja main()

Do tej pory we wszystkich programach istniała funkcja main(). Po co tak właściwie ona jest? Otóż jest to funkcja, która zostaje wywołana przez fragment kodu inicjującego pracę programu. Kod ten tworzony jest przez kompilator i nie mamy na niego wpływu. Istotne jest, że każdy program w języku C musi zawierać funkcję main().

Istnieją dwa możliwe prototypy (nagłówki) omawianej funkcji:
int main(void);
int main(int argc, char **argv); [3]

Argument **argc** jest liczbą nieujemną określającą, ile ciągów znaków przechowywanych jest w tablicy argv.
Wyrażenie **argv[argc]** ma zawsze wartość NULL.
Pierwszym elementem tablicy **argv** (o ile istnieje jest nazwa programu czy komenda, którą program został uruchomiony.
Pozostałe przechowują argumenty podane przy uruchamianiu programu.

Zazwyczaj jeśli program uruchomimy poleceniem:
program argument1 argument2
to argc będzie równe 3 (2 argumenty + nazwa programu) a argv będzie zawierać napisy program, argument1, argument2 umieszczone w tablicy indeksowanej od 0 do 2.

Przykładowy program, który wypisuje to, co otrzymuje w argumentach argc i argv:

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char **argv) {
  int i;
  for (i = 0; i<argc; ++i) {
    printf("%s", argv[i]);
  }
  return EXIT_SUCCESS;
}
```

Ostatnią rzeczą dotyczącą funkcji main jest zwracana przez nią wartość.
Jedynymi wartościami, które znaczą zawsze to samo we wszystkich implementacjach języka są 0,
EXIT_SUCCESS i EXIT_FAILURE zdefiniowane w pliku nagłówkowym stdlib.h.
Wartość 0 i EXIT_SUCCESS oznaczają poprawne zakończenie programu (co wcale nie oznacza, że makro EXIT_SUCCESS ma wartość zero), natomiast EXIT_FAILURE zakończenie błędne.
Wszystkie inne wartości są zależne od implementacji.
Jak zwrócić kilka wartości?

Generalnie możliwe są dwa podejścia: jedno to "upakowanie" zwracanych wartości – można stworzyć tak zwaną **strukturę**, która będzie przechowywała kilka zmiennych.
Prostszym sposobem jest zwracanie jednej z wartości w normalny sposób a pozostałych jako parametrów.

Przekazywanie parametrów
Gdy wywołujemy funkcję, wartość argumentów, z którymi ją wywołujemy, jest kopiowana do funkcji.
Kopiowana to znaczy, że nie możemy normalnie zmienić wartości zewnętrznych dla funkcji zmiennych.
W C argumenty są przekazywane przez wartość, czyli wewnątrz funkcji operujemy tylko na ich kopiach.
Możliwe jest modyfikowanie zmiennych przekazywanych do funkcji jako parametry - ale do tego w C potrzebne są wskaźniki.

Tablice jako parametr funkcji
Ponieważ nazwa tablicy jest wskaźnikiem do jej pierwszego elementu to możemy korzystać z tablic w ten sposób

```
#include<stdio.h>
 void czytaj (int c[],int i)
  {
   int j;
   for(j=0;j<i;j++)     scanf("%d",&c[j]);
   fflush(stdin);
  }

  void wyswietl (int d[],int i)
  {
   int j;
   for(j=0;j<i;j++)  printf("%d ",d[j]);
   printf("\n");
  }

  int main()
  {
   int a[5];
   printf ("Wprowadź 5 elementów listy \n");     czytaj (a,5);
   printf("Elementami listy są : \n");     wyświetl (a,5);
   return 0;
  }
```

Funkcje nie tylko mają dostęp do tablicy, ale i mogą ją zmieniać.

Programy

Pole prostokata

```
/* polpr1a.c  Pole prostokata */
/* Dyrektywy */
#include <stdio.h>
#include <conio.h>
```

```
#include <math.h>
/* Bliblioteka math.h - np. M_PI */
/* Makrodefinicja */
#define PI 3.141593

/* funkcja glowna main()*/
int main()
{
float a, b, p;            // deklaracje zmiennych
puts("Program polpr1a.c\n");
puts("Obliczenie pola prostokata: dane dlugosci a i b\n");
printf("PI=%lf (do innych obliczen, np. kola)",PI);
printf("\n\nPodaj a (0 - koniec)  => " );
scanf("%f",&a); /* wczytanie a */
while (a > 0)
 { // while
            printf("Podaj b lub 0 gdy b=a => " );
    scanf("%f",&b); /* wczytanie b */
    if (b==0)b=a;
            p=a*b;  /* pole */
            printf("Pole prostokata o bokach " );
            printf("%.3f i %.3f = %7.3f", a, b, p);
            printf("\n\nPodaj a (0 - koniec)  => " );
    scanf("%f",&a); /* wczytanie a */
  } // while

  // getch();
   return 0; /* powrot do systemu - main() zwraca 0 */
}
```

```
/* polpr1a.c  Obliczenie pola prostokata, zapis do pliku */
#include <stdio.h>
#include <conio.h>
#include <math.h>
#define PI 3.141593
/* Plik wynikow */
#define WYN "pprost.txt"

int main()
{
float a, b, p;
int nr=0;
char nazwisko[20]; /* nazwisko obliczajacego */
FILE *fp; /* zmienna plikowa */

puts("Obliczenie pola prostokata: dane dlugosci a i b\n");
printf("Wyniki zapisywane w pliku ");
puts(WYN);
printf("PI=%lf (do innych obliczen)",PI);
printf("\n\nPodaj a (0 - koniec)  => " );
scanf("%f",&a); // wczytanie a
// otwarcie pliku z kontrola
 if ( (fp = fopen(WYN, "w"))==NULL    )
  {
            puts("Nie mozna otworzyc pliku do zapisu");
            exit(1);
  }
  fprintf(fp, "Obliczenie pola prostokata o bokach a i b\n");
  fprintf(fp, "Lp     a        b        pole  \n");

while (a > 0)
 { // while
    nr++;
    fprintf(fp,"%3d  %6.3f",nr, a); // wydruk do pliku
```

```
            printf("Podaj b lub 0 gdy b=a => " );
    scanf("%f",&b); // wczytanie b
    if (b==0)b=a;
    fprintf(fp,"%10.3f",b);
            p=a*b;  // pole
            printf("Pole prostokata o bokach " );
            printf("%.3f i %.3f = %7.3f", a, b, p);
            fprintf(fp,"%10.2f\n",p);
            printf("\n\nPodaj a (0 - koniec)  => " );
    scanf("%f",&a); // wczytanie a
  } // while


  printf("\nNazwisko obliczajacego: ");
  fflush(stdin);
  gets(nazwisko); // wczytanie nazwiska
  fprintf(fp,"\nObliczyl:   %s", nazwisko);
  fclose(fp); // zamkniecie pliku

  // getch();
   return 0;
}
```

```
// polpr1.cpp  Pole prostokata CPP Dev C++
#include <cstdlib>
#include <iostream>
#include <math.h>
using namespace std;

int main(int argc, char *argv[]) // funkcja glowna
{
float a, b, p;
cout << "Program polpr1.cpp" << endl << endl;
cout << "Obliczenie pola prostokata: dane dlugosci a i b" << endl << endl;
cout << "Podaj a (0 - koniec) => " ;
cin >> a; // wczytanie a
while (a > 0)
 { // while
            cout << "Podaj b => "; // napis
    cin >> b;  // wczytanie b
            p=a*b; // pole
            cout << "Pole prostokat o bokach " << a << " i "  \
            << b << " = "<< p << endl << endl;
            cout << "Podaj a (0 - koniec) => " ;
            cin >> a;  // wczytanie a
  } // while

system("PAUSE");
return EXIT_SUCCESS;
}
```

Pola podstawowych figur płaskich

```
/*  pola_1.cpp  Obliczenie Pol figur plaskich. Menu z do – wybór typu char */
#include <stdio.h>
#include <conio.h>
// include <cstdlib>
// #include <iostream>
#include <math.h>
#define WYS2 puts("\n\n")
// using namespace std;

// Deklaracje funkcji
void poleprost();
void poletrojk();
```

```
void poletrapez();
double polekola();


int main(int argc, char *argv[])
{
 int wybor;
 float p;
 do
 {
            putchar('\n');
            puts("Program do obliczenia pol figur");
   puts("0. Koniec programu");
            puts("1. Pole prostokata");
            puts("2. Pole trojkata");
            puts("3. Pole trapezu");
            puts("4. Pole kola");
            puts("5. Zakoncz program");
            wybor=getchar();
            switch (wybor)
            {
                        case '1': WYS2; poleprost(); WYS2; fflush(stdin);  break;
                        case '2': WYS2; poletrojk(); WYS2; fflush(stdin);  break;
                        case '3': WYS2; poletrapez(); WYS2; fflush(stdin); break;
                        case '4': WYS2; p=polekola();
                        printf("Pole kola = %lf", p); WYS2; fflush(stdin); break;
                        case '5': case '0':  break;
                        default: puts("Niepoprawny wybor, powtorz\n");
            }
 } while ( (wybor != '5') && (wybor != '0'));
  // getch();
   return 0;
}

// Definicje funkcji
void poleprost()
{
 float a, b, p;
 puts("Obliczenie pola prostokata");
 printf("Podaj dlugosci bokow: a i b => "); scanf("%f %f", &a, &b);
 p=a*b;
 printf("Pole prostokata o bokach %f i %f = %f",a,b,p);
}

void poletrojk()
{
 float a, h, p;
 puts("Obliczenie pola trojkata");
 printf("Podaj dlugosci: a i h => "); scanf("%f %f", &a, &h);
 p=a*h/2.0;
 printf("Pole trojkata o boku %.3f i  wysokosci %.3f = %.3f",a,h,p);
 }

void poletrapez()
{
 float a, b, h, p;
 puts("Obliczenie pola trapezu");
 printf("Podaj dlugosci: a i b oraz h => "); scanf("%f %f %f", &a, &b, &h);
 p=0.5*(a+b)*h;
 printf("Pole trapezu o boku %.3f i  wysokosci %.3f = %5.3f",a,h,p);
 }

double polekola() // zwraca watosc
{
 double r, p, p1;
 double PI=4.0*atan(1.0),  pi = 4.0 * atan(1.0);
 puts("Obliczenie pola kola\n");
```

```
 printf("M_PI %lf  PI= %lf  pi = %lf",M_PI, PI, pi);
 printf("\nPodaj promien r  => ");
 scanf("%lf", &r);
 p=M_PI*r*r;
 p1=PI*r*r;
 printf("\nPole kola o promieniu %lf = M_PI*r*r= %lf",p);
 printf("\nPole kola o promieniu %lf = PI*r*r  = %lf",p1);
 printf("\n");
 return(p);
}
```

/* pola_1a.cpp Obliczenie Pol figur plaskich. Menu z do –wybór typu int */

```
#include <stdio.h>
#include <conio.h>
// include <cstdlib>
// #include <iostream>
#include <math.h>
#define KON puts("\n\n\n")
// using namespace std;

void poleprost();
void poletrojk();
void poletrapez();
float polekola();


int main(int argc, char *argv[])
{
 int wybor;
 float p;
 do
 {
            putchar('\n');
            puts("Program do obliczenia pol figur");
            puts("1. Pole prostokata");
            puts("2. Pole trojkata");
            puts("3. Pole trapezu");
            puts("4. Pole kola");
            puts("5. Zakoncz program");
            scanf("%d",&wybor);
            switch (wybor)
            {
                        case 1: poleprost(); KON; break;
                        case 2: poletrojk(); KON;  break;
                        case 3: poletrapez(); KON; break;
                        case 4: p=polekola();
                        printf("\nPole kola = %f", p);  KON;                  break;
                        case 5: case 0:  break;
                        default: puts("Niepoprawny wybor, powtorz\n");
            }
 } while ( (wybor != 5) && (wybor != 0));


  // getch();
   return 0;
}


void poleprost()
{
 float a, b, p;
 puts("Obliczenie pola prostokata");
 printf("Podaj dlugosci bokow: a i b => "); scanf("%f %f", &a, &b);
 p=a*b;
 printf("Pole prostokata o bokach %f i %f = %f",a,b,p);
}
```

```
void poletrojk()
{
 float a, h, p;
 puts("Obliczenie pola trojkata");
 printf("Podaj dlugosci: a i h => "); scanf("%f %f", &a, &h);
 p=a*h/2.0;
 printf("Pole trojkata o boku %.3f i  wysokosci %.3f = %.3f",a,h,p);
 }


void poletrapez()
{
 float a, b, h, p;
 puts("Obliczenie pola trapezu");
 printf("Podaj dlugosci: a i b oraz h => "); scanf("%f %f %f", &a, &b, &h);
 p=0.5*(a+b)*h;
 printf("Pole trapezu o boku %.3f i  wysokosci %.3f = %5.3f",a,h,p);
 }


float polekola()
{
 float r, p;
 puts("Obliczenie pola kola");
 printf("Podaj promien r  => "); scanf("%f", &r);
 p=M_PI*r*r;
 printf("Pole kola o promieniu %.3f  = %6.3f",p);
 return(p);
}
```