

Podstawy języka "C"

Podstawowe informacje o tej prezentacji

1. W tej prezentacji wszystkie przedstawione kody są dla środowiska 'Dev C++' - najbardziej popularnego, wyjątkowo łatwego do zainstalowania z Internetu i obsługi. Do tej prezentacji zupełnie wystarczy.
2. Aby poniższe kody przyjmowały symbole polskich liter, należy do każdego programu wpisać poniższe pogrubione dwie linie (w tej prezentacji nie dodawałem ich aby nie przedłużać kodów):

```
#include <stdio.h>
#include <locale.h>           // dla 'setlocale()'

int main()
{ setlocale(LC_CTYPE, "Polish"); // polskie znaki
  . . .
```

3. Jeżeli nie wiesz skąd biorą się już zdefiniowane funkcje - poznajemy je po nawiasach () zaraz po ich nazwach, np. *printf()* - lub wyrażenia nie deklarowane jawnie w programie, to na pewno znajdziesz je poniżej w opisie 'plików nagłówkowych' (*headers*) z przykładami ich zastosowań. To ważne aby wiedzieć, które z nich przyjmują argumenty a jeżeli je przyjmują to ile, jakiego typu one są i co one znaczą.
4. Pamiętaj: *wartość_zmiennej* dla znaku piszemy w ciapkach, np. 'W',
wartość_zmiennej dla ciągu znaków piszemy w cudzysłowie, np. "Ala",
wartość_zmiennej dla liczb piszemy bez ograniczników, np. 123.45 .

Podstawowe informacje o języku "C"

Tylko teoria:

Schemat każdego programu w języku "C" wygląda następująco:

```
#include<stdio.h>           // tym samym dołączamy podstawową bibliotekę języka C
...                         // <-- miejsce dołączenia innych 'bibliotek'
                           //      i ewentualnie nazw symbolicznych (patrz poniżej)
miejsce deklaracji zmiennych globalnych (widzianych w całym programie)
w tym ewentualnie 'prototypów' Twoich własnych funkcji

int main()                 // <-- musi być w każdym programie, obojętnie w którym miejscu
{                           // uruchomiony program szuka jej, bo zaczyna pracę od tej funkcji
    miejsce deklaracji zmiennych lokalnych (widzianych tylko wewnątrz tej funkcji)
    miejsce dla kodu programu
}
```

tu mogą być Twoje własne funkcje wywoływane np. z funkcji 'main' o schemacie takim samym jak podałem w funkcji 'main'

– Komentarze:

// Tu będzie opis programu.		← dla całej jednej linii lub
printf("Hello World !");	// drukuj	← dla objaśnienia pojedynczej linii
/* Tu będzie opis programu i może coś więcej */		← dla więcej niż jednej linii

– **Deklaracje zmiennych:** To nie tylko wprowadzenie nazwy zmiennych ale także przydzielenie im odpowiednich typów dla każdej zmiennej osobno (char, int, float, double) co powoduje automatyczne przedzielanie im rozmiaru pamięci:

- Np. **char** bierze 1 bajt (*byte*) pamięci
int bierze 2 bajty (*small int*) lub 4 bajty (*long int*) pamięci w *Turbo C++*; w *Dev C++* może być więcej
float bierze 4 bajty pamięci w *Turbo C++*; w *Dev C++* może być więcej bajtów
.... (itd.)
- Przedstawienie ich w **printf()** – tu są różne możliwości: po % może wystąpić: *d* lub *i* (dla całkowitej liczby dziesiętnej), *o* (dla liczby w systemie ósemkowym), *x* (dla liczby w systemie szesnastkowym), *u* (dla liczby bezznakowej, *unsigned*), *c* (dla znaku), *s* (dla łańcucha znaków), *f* (dla liczby zmiennoprzecinkowej), *e* (dla notacji wykładniczej – np. 2E05 lub 2e05 oznacza 200000), *g* (dla notacji wykładniczej o wykładniku < -4), *p* (dla wskaźnika, *pointer'a*).
- char ile[5] = "12345" nie jest liczbą i dodać 1 do niej się nie da, ale można ją zamienić na liczbę instrukcją **atoi** (**A**lphabetic **T**O Integer) i ten łańcuch znakowy stanie się liczbą (patrz: "Pliki nagłówkowe" → stdlib.h).
- 'rzut' (*casting*) – to jawne wymuszanie typu zmiennej w środku programu. Np. sqrt(**(double)**n) – jak *n* było 'int', to tu zmeni swój typ na 'double'.

– **Definiowanie nazw symbolicznych** (*symbolic parameters*) (lepszą nazwą to: stałe symboliczne – *constant symbolic parameters*), np. #define MAX 100

Taka deklaracja pozwala używać słowa MAX w miejsce 100 w wielu miejscach w programie. Wymiana jej na, np.

```
#define MAX 1000
```

automatycznie wymieni wszystkie wystąpienia MAX w programie na 1000.

Liczba 'pi' już jest zdefiniowana w "header'ze" math.h jako M_PI (patrz: "Pliki nagłówkowe" → math.h) z bardzo dużą precyzją, ale deklaracja:

```
#define PI 3.14159265
```

nie jest błędem choćbyśmy nazwali ją M_PI zamiast PI. Tak deklarowana zmienna staje się bardziej jawna.

– **Zmienne lokalne (wewnętrzne) i globalne (zewnętrzne).** Funkcje natomiast, zawsze są globalne (zewnętrzne).

- Uwaga: wszystko jest w środku programu, ale zmienne wewnętrzne (lokalne) są definiowane wewnątrz jakiejś funkcji i nie są widziane poza nią. Zmienne 'zewnętrzne' to inne, definiowane poza funkcjami, najczęściej tuż na początku kodu, u nas tuż przed funkcją **main**, która zawsze powinna być (ale nie musi) jako funkcja pierwsza. Często spotyka się funkcję **main** jako ostatnią w programie - wtedy nie trzeba określać prototypu każdej innej funkcji, ale zastanawia, czy to wystarczający powód aby coś pierwszego było w kodzie ostatnim.
- Zauważ, że jeżeli zaczynamy coś od # to nie ma tu ani znaku przypisania (=) ani nie kończymy tego zapisu znakiem średnika (;) .

– **Biblioteki C** (*C libraries*) – trzeba je dołączać poprzez instrukcję (tu: 'dyrektywę') **#include**. Np. **#include <stdio.h>**

- Patrz: "Pliki nagłówkowe", aby wiedzieć do czego służą.

– Operatory:

- Operatory arytmetyczne: +, -, *, /, % (modulo, reszta z dzielenia), ++, --, +=, -=, *=, /=, %=.
 - Operatory relacji i logiczne.
 - Operatory relacji (<, <=, >, >=, ==, !=) ← '==' to prawdziwy znak 'równości'.
- Np. **if(!zle)** i niech **zle = 0** (znaczy: fałsz). Teraz **!zle != 0** (znaczy: prawda) co umożliwia wykonanie instrukcji w klamrach {}, tuż po zapisie **if(!zle)**. **if(0)** unieruchamia te instrukcje. Zapis prowadzący do **if(0) {...} else {instrukcje dla 'else'}** powoduje wykonanie tylko instrukcji dla 'else'.

➤ Operatory logiczne

&&	– zwany operatorem AND	'i'
	– zwany operatorem OR	'lub'
!	– zwany operatorem NOT	'nie' (zaprzeczenie tego co następuje po !)

- Operatory bitowe (&, |, ^, <<, >>, ~) to operacje na bitach.

Tu nie będziemy się zagłębiać, ale patrz: "Tablice języka C", gdzie są wystarczająco dobrze opisane.

- Operatory przypisania (tam, gdzie jest pojedynczy znak =. Wartość prawej strony przypisujemy zmiennej po lewej stronie znaku =):

Czy wiesz co to znaczy: `i++`, `++i`, `i--`, `--i` albo `i += 2` ? Patrz poniżej:

➤ Operatory dwuargumentowe: `+`, `-`, `*`, `/`, `%`, `<<`, `>>`, `&`, `^`, `|` (poniżej 'wyr' jak 'wyrażenie')

Np. <code>wyr1 operator= wyr2</code>	\longleftrightarrow	<code>wyr1 = wyr1 operator wyr2</code>	(zapisy równoważne sobie)
<code>x *= y + 1</code>	\longleftrightarrow	<code>x = x * (y + 1)</code>	(zapisy równoważne sobie)

➤ Operatory jednoargumentowe (*unary operators*): `++`, `--`

<code>i++</code>	– znaczy to samo co	<code>i = i + 1</code> (dodaj 1 do zmiennej 'i')
<code>++i</code>	– znaczy to samo co	<code>i = i + 1</code> (dodaj 1 do zmiennej 'i' ale zrób to po ważniejszej operacji)
<code>i--</code>	– znaczy to samo co	<code>i = i - 1</code> (odejmij 1 od zmiennej 'i')
<code>--i</code>	– znaczy to samo co	<code>i = i - 1</code> (odejmij 1 od zmiennej 'i' ale zrób to po ważniejszej operacji)

Zwykle nie ma różnicy przy zastosowaniu `++` czy `++i` (podobnie powiemy o `i--` czy `--i`).

(Oprócz nich są bitowe operatory przypisania: `<<=`, `>>=`, `&=`, `^=`, `|=` ← tu nie będziemy się zagłębiać.)

- Inne (pozostałe) operatory

<code>sizeof()</code>	– podaje rozmiar zajmowanej pamięci czegoś, co jest w nawiasie.
<code>&</code>	– podaje adres (numer komórki pamięci) zmiennej , która jest zaraz za znakiem <code>&</code> .
<code>*</code>	– wskaźnik (pointer) do zmiennej . Trzyma w sobie ten własny numer komórki pamięci.
<code>?:</code>	– wyrażenie warunkowe – jeżeli warunek jest prawdą, to przyjmij za wartość zmiennej po lewej stronie znaku '=' wartość pierwszej zmiennej w wyrażeniu; w przeciwnym razie przyjmij wartość drugiej zmiennej.

Np.: `z = (a > b) ? a : b` to wybieranie maksymalnej liczby.

Jeżeli `z = (a > b)` jest prawdą (`a > b`) to `z = a` ('z' będzie miało wartość taką jak 'a')

Jeżeli `z = (a > b)` jest fałszem (`a <= b`) to `z = b` ('z' będzie miało wartość taką jak 'b')

- Oprócz 'typów' zmiennych i 'operatorów' pomiędzy zmiennymi (i stałymi jak w: `2*x`), mamy wyrażenia – każde 'określenie' to właśnie 'wyrażenie'. Np. `char linia[30]` – sugeruje linię o 30 znakach (złożonych np. z '_____').

– **Tablice (arrays)**. Np. `char tab[30]` – tablica 30-to elementowa złożona ze znaków (nie cyfr) – w `printf` będzie to `%s` dla łańcucha znaków a `%c` dla pojedynczego znaku.

- Wskaźniki (*pointers*) – akurat w "C" to ważna rzecz; patrz poniżej w tej prezentacji.

– **Funkcje** – najważniejsza to: (z czym wychodzi) `main` (co pobiera).

- Jeżeli od niej (funkcji `main`) się zaczyna i na niej (`main`) się kończy, to zwykle ona 'z niczym nie wychodzi' i 'nic nie pobiera': **(`!`)`main`(`!`)** lub **(`void`)`main`(`void`)**. `void` znaczy 'nic/nieważne/puste'.
- Inne funkcje muszą coś pobrać i zwykle coś oddają (nie oddają wtedy, gdy na nich kończy się proces, ale np. może być wywołana funkcja, która 'przy okazji' kopiuje lub czyta łańcuch tekstowy – te funkcje mogą być bezargumentowe).
- Deklarowanie/prototyp funkcji (*function prototyping*) – następuje tak, jak deklarowanie zmiennych.
- Przekazywanie argumentów (*passing values*, z wywołania funkcji) do jej parametrów (zdefiniowanej funkcji):
 - bezpośrednio wartościami zmiennych (to są właśnie argumenty) – *passing by values*,
 - przez referencje (adresy pod którymi są te zmienne) – *passing by references*.
- **return wyrażenie**; – pamiętaj: Każda funkcja może zwracać (do miejsca, z którego jest wywoływana) jedną wartość. Może zwrócić jednak całą strukturę (przecież to jedna rzecz) zawierającą wartości mnóstwa zmiennych.

– **'Definicja' a 'Deklaracja' zmiennej.**

- Definicja zmiennej – to miejsce, gdzie zmienna jest tworzona i rozmiar jej pamięci przydzielony.
- Deklaracja zmiennej – to miejsce, gdzie określa się naturę zmiennej ale nie przydziela się jej pamięci, np. `char tablica[];` .

Instrukcje decyzji (wyboru)

Teoria:

- **if**(warunek) [**else**] ← 'else' może nie występować w instrukcji **if** – nie zapomnij o nawiasach { i }.
 if może wystąpić pod innym **if** (zagnieżdżanie instrukcji **if**).
- **switch** – w **switch** zawsze występują **cases** (przypadki).
 W tej instrukcji (**switch**) zawsze pisz '**break**' pod koniec każdego '**case**'.
- Sortowanie: Który znak jest 'większy' a który 'mniejszy' – ASCII numeruje znaki i te numery są sortowane.

if

if ... else ...
zagnieżdżone if

switch case break

zagnieżdżone switch

? : operator

Praktyka:

if (warunek, który jest prawdą) {instrukcje}

Uwaga: **if** (liczba != 0) jest tym samym co **if** (liczba) ponieważ instrukcja **if** jest wykonywana, gdy warunek w nawiasie nie przyjmuje wartości 0 (zera).

if (liczba > 5) - jeżeli wartość liczby jest większa od 5 to w nawiasie jest logiczna prawda z wartością różną od zera. W przeciwnym wypadku wartość w nawiasie jest zerem co czyni wyrażenie logicznym fałszem.

Jeżeli w bloku pomiędzy { i } jest tylko jedna instrukcja zakończona znakiem ; to nawiasy klamrowe można pominąć.

```
// Test na parzystość liczby
#include <stdio.h>

int main()
{
    int liczba;
    printf("Tu wpisz liczbę całkowitą: ");
    scanf("%d", &liczba);

    if (liczba % 2 == 0)    // tu znak % mówi: pokaż resztę z dzielenia liczby przez 2
        printf("\n Liczba %d jest liczbą parzystą \n", liczba);
    else
        printf("\n Liczba %d jest liczbą nieparzystą \n", liczba);

    return 0;
}
```

switch (zmienna) { **case** wartość zmiennej : instrukcje; ... **break**; ... **default**: ...;}

Najpierw zagnieżdżenie **if**'ów zamiast funkcji **switch()**:

```
// Wybieranie jednego z wielu - zagnieżdżenie wyrażenia 'if'
#include <stdio.h>
#include <conio.h>          // dla getch()
#include <ctype.h>          // dla toupper()

int main()
{
    char wykres;
    printf("\n Potrzebne wykresy: Kołowy, Słupkowy, Punktowy, Liniowy, Trójwymiarowy");
    printf("\n\n Wprowadź pierwszą literę nazwy wykresu, który chcesz otrzymać: ");
    wykres = toupper( getch() );    // jeżeli wprowadzisz małą literę to będzie ona
    printf("\n\n");                //      zamieniona na wielką

    if (wykres == 'K')
        printf("Tu będzie wykres kołowy \n");
    else if (wykres == 'S')
        printf("Tu będzie wykres słupkowy \n");
    else if (wykres == 'P')
        printf("Tu będzie wykres punktowy \n");
    else if (wykres == 'L')
        printf("Tu będzie wykres liniowy \n");
    else if (wykres == 'T')
        printf("Tu będzie wykres trójwymiarowy \n");
    else printf("Nieważny wybór. \n");

    return 0;
}
```

Powyższy kod można zastąpić bardziej czytelnym:

```
// Wybieranie jednego z wielu - instrukcja 'switch'
#include <stdio.h>
#include <conio.h>          // dla getch()
#include <ctype.h>          // dla toupper()

int main()
{
    char wykres;
    printf("\n Potrzebne wykresy: Kołowy, Słupkowy, Punktowy, Liniowy, Trójwymiarowy");
    printf("\n\n Wprowadź pierwszą literę nazwy wykresu, który chcesz otrzymać: ");
    wykres = toupper(getche());    // jeżeli wprowadzisz małą literę to będzie ona
    printf("\n\n");                //      zamieniona na wielką

    switch (wykres)
    {
        case 'K': printf("Tu będzie wykres kołowy \n"); break;
        case 'S': printf("Tu będzie wykres słupkowy \n"); break;
        case 'P': printf("Tu będzie wykres punktowy \n"); break;
        case 'L': printf("Tu będzie wykres liniowy \n"); break;
        case 'T': printf("Tu będzie wykres trójwymiarowy \n"); break;
        default : printf("Nieważny wybór. \n");
    }

    return 0;
}
```

- Uwaga:** 1. `getche()` pokazuje wartość przyciśniętego klawisza; `getch()` tego nie robi.
2. 'break' jest konieczny po każdym 'case'. Bierze się to stąd, że pojedynczy 'case' może obsługiwać wiele instrukcji - raz uchwycony czytałby wszystko do końca z 'default' włącznie. Także jedna instrukcja (lub ich zestaw) może obsługiwać wiele 'cases' jak w poniższym przykładzie.
- Patrz poniżej: 'Instrukcje kontrolne pętli'.

```
// Jedna instrukcja obsługująca wiele przypadków (cases)
#include <stdio.h>

int main()
{   int cyfra;
    printf("\n Wpisz cyfrę większą od 1: ");
    scanf("%d", &cyfra);
    printf("\n\n");

    switch (cyfra)
    {   case 2:
        case 4:
        case 6:
        case 8:
            printf("Cyfra %d jest parzysta \n", cyfra); break;
        case 3:
        case 5:
        case 7:
        case 9:
            printf("Cyfra %d jest nieparzysta \n", cyfra); break;
        default : printf("Nieważny wybór. \n");
    }

    return 0;
}
```

? : operator czyli 'wyrażenie warunkowe'

Wyrażenie1 ? Wyrażenie2 dla prawdy Wyrażenia1 : Wyrażenie3 dla fałszu Wyrażenia1

```
// Wyrażenie warunkowe
// Tutaj, złożenie dwóch wyrażeń warunkowych (zagnieżdżenie wyrażeń warunkowych):
// W miejscu 'Wyrażenia3' występuje następne całe wyrażenie warunkowe.
#include <stdio.h>

int main()
{   int rok;
    printf("Podaj rok pomiędzy 1582 a 4500: "); // to przedział ważności kalendarza
    scanf("%d", &rok);
    (rok % 4 == 0 && rok % 100 != 0) ? printf("%d to rok przestępny", rok) : (rok %
400 == 0) ? printf("%d to rok przestępny", rok) : printf("%d to zwykły rok", rok);

    return 0;
}
```

Pętle

Teoria:

- Instrukcja **for** – pętla dla przewidywanej ilości kroków pętli.
 - **for(od; aż do fałszu; przyrost/krok)**. Np. `for (i = 0; i < n; i = i+1)` ← rób od $i=0$ dla warunku $i < n$, za każdą nową pętlą zwiększ i o 1. Już wiemy, że będzie n kroków pętli, bo zanim i dojdzie do n , będzie n takich kroków.
 - Każdy zwarty zestaw instrukcji musi być ujęty w nawiasy klamrowe `{ }`, chyba, że ten 'zestaw' to tylko jedna instrukcja – jedna linia zakończona średnikiem `;`.
 - **Instrukcje warunkowe i instrukcje bezwarunkowe** – jak je rozróżnić:
Jeżeli program zawsze natrafi na daną instrukcję to musi ją wykonać – jest to instrukcja bezwarunkowa.
W **if**ie są instrukcje warunkowe – zależne od warunków ustalonych przez instrukcję **if**.
- Instrukcje **while** i **do ... while**
 - **while**(wykonuj kiedy spełniony jest warunek) ← tutaj może nie być wykonana żadna pętla (**while**)
 - **do ... while**(wykonuj kiedy spełniony jest warunek) ← musi być wykonana przynajmniej jedna pętla (jeden krok **do ... while**)
- Wcześniejsze (niż to 'chce' **for** lub **while**) wyjście z pętli instrukcjami
 - **break** – natychmiastowe wyjście z pętli. W przypadku zagnieżdżenia, wychodzi z tej pętli, w której akurat jest
 - **continue** – przerwanie tego kroku (tej pętli) w celu rozpoczęcia następnej pętli (pętle są kontynuowane, ale niektóre niepełnie)
- Instrukcja **goto nazwa_etykiety** – skocz do etykiety o podanej nazwie (etykieta to po angielsku: *label*)
Np. **goto** problem;
Gdzieś w programie na początku jakiejś linii jest słowo *problem:* ('problem' i dwukropek)
i tam skoczy przetwarzanie programu. Poza takim skokiem z **goto**, kompilator ignoruje to słowo.

Praktyka:

for (zaczynij od ; wykonuj, aż ten warunek będzie fałszem ; skok zmiennej) { ... }

Uwaga: 'zaczynij od' może być pojedynczym warunkiem lub zespołem warunków odseparowanych przecinkiem. Brak zapisu oznacza wszystkie przypisania do tej pory napotkane.
'wykonuj, aż ten warunek będzie fałszem' może być złożonym warunkiem.
'skok zmiennej' może obejmować więcej niż jedną zmienną.

Przykłady:

```
for(int i = 1; i < 3; i++)  
lub  
int i;  
for(i = 1; i < 3; i++)  
lub  
int i = 1;  
for( ; i < 3; i++)  
lub  
for (int i = -3, j = 0, k = 10; i < j && i < k; i++, k=k-2)
```

```
// Złożona pętla 'for'  
#include <stdio.h>  
  
int main()  
{  
    for (int i = -3, j = 0, k = 10; i < j && i < k; i++, k=k-2)  
        printf("i = %d, j = %d, k = %d \n", i, j, k);  
  
    return 0;  
}
```

while (wykonuj, gdy wyrażenie jest prawdą) {...}

```
#include <stdio.h>

int main()
{   int ocena = 6;

    while (ocena)                // lub, w tym wypadku: while (ocena > 0)
    {   ocena--;
        printf("\n");            // ustaw kursor w nowej linii
        printf(" Wartość 'ocena' = %d", ocena);
        if (ocena < 2)
            printf(" - nie ma takiej oceny w szkole");
        // zamiast powyższej linii wstaw jedną z następujących:
        // continue;            // patrz poniżej: 'Instrukcje kontrolne pętli'
        // break;               // patrz poniżej: 'Instrukcje kontrolne pętli'
        // goto to_koniec;      // patrz poniżej: 'Instrukcje kontrolne pętli'
    }

    printf("\n Wartość 'ocena' po wyjściu z pętli 'while' = %d\n", ocena);
    to_koniec:
    printf("\n Tu jest koniec programu");

    return 0;
}
```

do {...} while (wykonuj, gdy wyrażenie jest prawdą);

Typowym zastosowaniem pętli 'do ... while();' jest wpisywanie hasła ponieważ pętla ta musi być procesowana przynajmniej raz.

```
#include <stdio.h>
#include <string.h>    // dla strcmp() - strings compare - porównanie łańcuchów znakowych
#include <conio.h>     // dla getch()

char haslo_wejscie[10]; // Tablica przygotowana do przyjęcia wpisywanego hasła
                        // hasło ma być nie dłuższe niż 10 znaków (to tylko przykład do wymiany)
char haslo[] = "Artur"; // Prawdziwe hasło - takie ma być wprowadzone (to tylko przykład do wymiany)
int dlugosc_hasla;      // Każdy wprowadzony znak zwiększa 'dlugosc_hasla' o 1
int i, j=0;             // 'i' - numer kolejnego znaku hasła (od 1)
                        // 'j' - ilość prób wpisania hasła

int main()
{
    do
    {
        if ( j > 0 && j < 3)
            printf("\n Niepoprawne hasło. Spróbuj ponownie");
        j++;
        if ( j > 3 ) break;

        printf("\n\n Wpisz hasło: ");

        i = 0;
        while( ( haslo_wejscie[i] = getch() ) != '\x0D' ) // aż do przyciśnięcia klawisza [ENTER]
        {   i++;
            printf("*"); // każdy wprowadzony znak zastępowany jest wizualnie przez gwiazdkę
                // ale 'haslo_wejscie' jest budowane i będzie porównane z wartością zmiennej 'haslo'
        }
        dlugosc_hasla = i;
        haslo_wejscie[dlugosc_hasla] = '\0'; // zamknięcie łańcucha znakowego

        printf("\n Hasło: %s, długość hasła: %d \n", haslo_wejscie, dlugosc_hasla); // tylko dla testu

    } while ( (strcmp(haslo_wejscie, haslo) != 0) && j <= 3 );
    // strcmp(str1, str2) == 0 gdy str1 jest identyczne z str2; długość hasła nie może być mniejsza niż 3
}
```



```

if ( j > 3 )
    printf("\n\n Trzykrotnie wprowadzono nieprawidłowe hasło - skontaktuj się z administratorem \n");
else
    printf("\n Hasło poprawne. Możliwość kontynuowania procesu \n");

return 0;
}

```

Wszystkie pętle można wzajemnie ze sobą zagnieżdżać.

Instrukcje kontrolne pętli

break; <-- napotkana, wychodzi z pętli wewnętrznej (dla **while**, **do**, **for** i **switch**).

continue; <-- napotkana, przerywa działania w pętli wewnętrznej i kontynuuje następny krok działania tej pętli (dla **while**, **do ... while** i **for**).

goto etykieta; <-- napotkana, powoduje przeskok do miejsca w programie, gdzie występuje słowo pod nazwą *etykieta* z dwukropkiem i może ona występować poza pętlą.
 i gdzieś poniżej w programie
etykieta:
 ...

Powyższe 'instrukcje kontrolne pętli' są instrukcjami skoku. Dodatkową instrukcją skoku jest '**return**' dla funkcji. Zwraca jeden obiekt do miejsca wywołania funkcji o typie takim, jakim deklarujemy zwrot w prototypie funkcji. Napotkana kończy działanie tej funkcji.

Funkcje

Funkcje wprowadza się po to, aby:

- zmniejszyć kod funkcji głównej (to znaczy tej, z której jest wywoływana),
- wywołana funkcja wykonywała się wielokrotnie według takiego samego schematu, na takiej samej ilości zmiennych wejściowych, tego samego typu.

Przykład:

```

// Przykład funkcji
#include<stdio.h>

int zmien(int);          /* deklaracja funkcji, prototyp funkcji */

int main()
{
    int a = 1;           /* deklaracja zmiennych lokalnych */
    int b = 3;
    printf("Przed zmianą: a = %d\n", a);
    a = zmien(a);        /* Wywołanie funkcji 'zmien' dla zmiennej 'a'
    printf("Przed zmianą: b = %d\n", b);
    b = zmien(b);        /* Wywołanie tej samej funkcji 'zmien' dla zmiennej 'b'
    printf("Po zmianie: a = %d\n", a);
    printf("Po zmianie: b = %d\n", b);
    return 0;
}

int zmien(int x)
{
    x += 3;
    return x;
}

```

```
}
```

Inny przykład:

```
// Inny przykład funkcji
#include <stdio.h>

int max(int num1, int num2); /* deklaracja funkcji, prototyp funkcji */

int main ()
{   int a = 100;                /* deklaracja zmiennych lokalnych funkcji main */
    int b = 200;
    int najwieksza;
    najwieksza = max(a, b);     /* wywołanie funkcji aby uzyskać wartość funkcji max */
    /* po wywołaniu funkcji w miejscu 'max(a,b)' będzie wartość zmiennej 'rezultat' */

    printf( "Największą z dwóch liczb %d i %d jest %d\n", a, b, najwieksza );

    return 0;
}

/* funkcja zwracająca wartość maksymalną dwóch liczb */
int max(int num1, int num2)
{   int rezultat;               /* deklaracja zmiennej lokalnej funkcji max */

    rezultat = (num1 > num2) ? num1 : num2;

    // Ostatnia linia powyżej zastępuje te cztery linie poniżej:
    //   if (num1 > num2)
    //       rezultat = num1;
    //   else
    //       rezultat = num2;

    return rezultat;
}
```

W pierwszych liniach programu deklarujemy funkcję (tworzymy tzw. 'prototyp funkcji'). W ten sposób wprowadzamy trzy elementy każdej funkcji z wszelkimi konsekwencjami z tym związanymi (ilość pamięci, sposób przekazania parametrów z programu do funkcji, ...):

Np. **int** *Moja_funkcja* (**char** a) Niech scenariuszem będzie zamiana znaku ASCII na liczbę

1. **int** – typ obiektu, jaki funkcja ma zwracać do miejsca, z którego była wywołana
2. *Moja_funkcja* – nazwa funkcji, jakąkolwiek nazwę wymyślisz
3. **char** a – typ i jego parametr przyjmujący wartość do funkcji, gdy funkcja jest wywoływana

Gdzieś w programie:

```
int liczba;
char znak = 'A';
...
liczba = Moja_funkcja(znak);    // Nazwa funkcji jest jej wywołaniem
...
```

Gdy funkcja zostanie wywołana i zwróci wartość w miejsce jej wywołania, to wyobraź sobie, że linia:

```
liczba = Moja_funkcja(znak);
```

zostaje zamieniona na (tego nie widać, ale tak jest):

```
liczba = 65;                                <-- 65 to liczba ASCII dla znaku 'A'
```

Argument funkcji (wartość zmiennej w wywołaniu funkcji nazywamy *argumentem* tej funkcji)

Są dwa sposoby przekazywania argumentów do pracującej (zdefiniowanej) funkcji, poprzez:

- wartość argumentu,
- referencje (czyli adres argumentu)

Przekazywanie argumentów do funkcji przez ich wartość:

```
// Przekazywanie argumentów funkcji przez ich wartość
#include<stdio.h>

int zmien1(int);
int zmien2(int);

int main()
{   int a = 1;
    int b = 3;

    printf("Przed zmianą: a = %d\n", a);
    a = zmien1(a);                // tu 'a' w (a) jest argumentem funkcji 'zmien1'

    printf("Przed zmianą: b = %d\n", b);
    b = zmien2(b);                // tu 'b' w (b) jest argumentem funkcji 'zmien2'

    printf("Po zmianie: a = %d\n", a);
    printf("Po zmianie: b = %d\n", b);
    return 0;
}

int zmien1(int x) // tu 'x' jest parametrem funkcji 'zmien1', przyjmującym wartość 'a'
{   x += 3;       // parametr (tu 'x') można traktować jak element symboliczny
    return x;
}

int zmien2(int y) // tu 'y' jest parametrem funkcji 'zmien2', przyjmującym wartość 'b'
{   y += 5;       // parametr (tu 'y') można traktować jak element symboliczny
    return y;
}
```

Wady tego (klasycznego) podejścia:

- kopiowanie *a* do *x* (w powyższym przykładzie także *b* do *y*)
- deklarowanie zmiennej *x* (w powyższym przykładzie także zmiennej *y*)
- ponieważ każda funkcja może zwracać wartość co najwyżej jednej zmiennej, trzeba wywołać tę samą funkcję ponownie dla osiągnięcia podobnego celu.

POCZĄTEK POINTERÓW

Przekazywanie argumentów przez referencje (adres komórki pamięci)

Zalety tego podejścia:

- nic nie jest kopiowane
- nie ma zwracania wartości jakiegokolwiek parametru:
 - modyfikowane są tylko wartości pod przekazanymi adresami;
 - w ten sposób możemy modyfikować wartość więcej niż jednej zmiennej.

```
// Przekazywanie argumentów funkcji przez adresy tych argumentów
#include<stdio.h>

void zmien(int *, int *);          // prototyp funkcji

int main()
{
    int a = 1;
    int b = 3;
    printf("Przed zmianą: a = %d\n", a);
    printf("Przed zmianą: b = %d\n", b);

    zmien(&a, &b);                // wywołuje funkcję, która nic nie zwraca
    printf("Po zmianie: a = %d\n", a);
    printf("Po zmianie: b = %d\n", b);
    return 0;
}

void zmien(int *ptr_a, int *ptr_b) // chwyta wartości pod adresami
{
    *ptr_a += 3;                  // i modyfikuje je pod tymi adresami
    *ptr_b += 5;                  // a adresy widziane są także w funkcji 'main'
}
```

Tablice w C i pointery (wskaźniki)

Co to jest adres obiektu?

Adres obiektu (&obekt) to numer pierwszej komórki pamięci (gdzie jego zapis się zaczyna) obiektu. Ponieważ znamy typ obiektu, znamy również jego długość w bajtach. Tak więc wartość obiektu możemy odczytać. Zamiast 'zmienna', piszę 'obekt' bo może być nim nie tylko liczba np. typu `int` ale np. cała struktura (patrz poniżej).

```
// Adres zmiennej/obektu
#include <stdio.h>

int main ()
{
    int  zmienna_1;
    char zmienna_2[10];

    printf("Adres zmiennej o nazwie zmienna_1: %x\n", &zmienna_1 );
    printf("Adres zmiennej o nazwie zmienna_2: %x\n", &zmienna_2 );
    return 0;
}
```

Co to są pointery (wskaźniki)?

Pointer to zmienna, której wartość jest adresem do innej zmiennej, to znaczy, że jest to bezpośredni adres do konkretnej komórki w pamięci komputera.

Ogólny zapis deklaracji zmiennej pointera:

```
typ *nazwa_zmiennej;
```

Można więc *pointer* deklarować tak (to tylko podstawowe jego typy):

```
int    *ip;           /* pointuje do integer - pointer typu integer (liczby całkowitej) */
double *dp;           /* pointuje do double - pointer typu double (liczby podwójnej precyzji) */
float  *fp;           /* pointuje do float - pointer typu float (liczby zmiennoprzecinkowej) */
char   *cp;           /* pointuje do character - pointer typu char (znakowego) */
```

W linii pierwszej **ip** jest pointerem (nie ***ip**). Dlaczego pointerem? Bo ma ***** przed sobą. Na razie taki *pointer* nigdzie nie 'pointuje'; to tylko deklaracja *pointera* jak zwykłej zmiennej.

Jak używać pointery (wskaźniki do numeru pierwszej komórki obiektu)?

```
// Pointer i adres pierwszej komórki pamięci zmiennej/obektu
#include <stdio.h>

int main ()
{
    int var = 20; /* deklaracja aktualnej zmiennej */
    int *ip;      /* deklaracja 'pointera'. Na razie jest to zwykła zmienna typu int */
    ip = &var;    /* ip trzyma adres zmiennej var */

    printf("Adres zmiennej var: %x\n", &var );

    /* adres pamięci w zmiennej pointera */
    printf("Adres przechowywany w pointerze (zmiennej) ip: %x\n", ip );

    /* dostęp do wartości poprzez pointer */
    printf("Wartość zmiennej *ip: %d\n", *ip );

    return 0;
}
```

ip = &var; <-- wreszcie wiemy, że **ip** pointuje (oczywiście do adresu pierwszej komórki) do zmiennej **var**.

Gdy już nie potrzebujemy 'pointować' do zmiennej 'var', ten sam pointer (tu: **ip**) możemy użyć do 'pointowania' do innego obiektu, oczywiście musi on być tego samego typu.

Wynik:

```
Adres zmiennej var: bffd8b3c
Adres przechowywany w pointerze (zmiennej) ip: bffd8b3c <-- oczywiście ten sam co powyżej
Wartość zmiennej *ip: 20
```

Skoro **ip = &var** a ***ip = 20** to z tego wynika, że ***(&var) = 20**.

Pointery (wskaźniki) są bardzo związane z tablicami. Na przykład:

```
// Pointery i tablice
#include <stdio.h>
```

```

int main(void)
{
    int a[5] = {7, 2, 9, 4, 6};

    // Deklarujemy zmienną typu int, która już wie, że będzie pointerem.
    // W przypadku tablic, od razu deklaruj pointer z gwiazdką: *ptr_a
    // Poniższy zapis powoduje, że ptr_a wskazuje do pierwszego elementu tablicy a
    // Może wskazywać inny element, np. trzeci: *ptr_a = &a[2]

    int *ptr_a;
    ptr_a = &a[0];      // teraz '*ptr_a' i 'a[0]' są tym samym obiektem

    // *ptr_a jest równy 7 (czyli wartość a[0] jest równa 7)
    *ptr_a = 10;        // zmiana wartości pierwszego elementu z 7 na 10

    printf("\n Wartość trzeciego elementu tablicy = %d\n", *(ptr_a + 2));

    // Skoro ptr_a wie, że jest pointerem do pierwszego elementu tablicy to
    // ptr_a + 1 będzie wskazywał do następnego elementu tablicy
    // Tak więc *(ptr_a + 1) jest wartością następnego elementu tablicy i jest
    // równy a[1]
    // W zapisie '*(ptr_a + 1)' 1 to niekoniecznie jeden bajt lecz tyle bajtów,
    // ile zawiera typ pointera ptr_a

    return 0;
}

```

Podobnie dla **char *ptr_a**

```

// Pointery i łańcuchy znakowe, które są przecież tablicami znaków
#include <stdio.h>

int main(void)
{
    char a[5] = {'A', 'r', 't', 'u', 'r'};      // albo: char a[] = "Artur";
    char *ptr_a;

    ptr_a = &a[0];      // teraz '*ptr_a' i 'a[0]' są tym samym obiektem
    printf("\n Wartość czwartego elementu tablicy = %c\n", *(ptr_a + 3) );

    return 0;
}

```

A tu wypisuje cały łańcuch znakowy, znak po znaku, przy pomocy **char *ptr_a**

```

#include <stdio.h>

int main()
{
    // daj szansę wpisania '\0' na koniec łańcucha, deklarując tu więcej niż 21
    char a[25]="Borland International";      // faktyczna długość łańcucha to 21

    char *ptr_a = &a[0];      // tu się kończy rola nazwy; przejmuje ją pointer
    printf("%c", *ptr_a);      // 'inicjalizujący' printf - dla pierwszego znaku
    while (*ptr_a++)
        printf("%c", *ptr_a); // pętla, aż do końca łańcucha znakowego
}

```

```
return 0;  
}
```

Operator * i tablice wielowymiarowe

W tym programie nie ma dosłownie zdefiniowanych pointerów. Jednakże zawsze musimy pamiętać, że początek każdego 'obiektu' ma swoją komórkę pamięci a ponieważ każdy 'obiekt' ma swój typ (type jak int, char itd.), to ma też ściśle określony rozmiar.

```
// Pointery i tablice wielowymiarowe  
#include <stdio.h>  
  
int main()  
{   int a[3][5] = { { 7,  2,  9,  4,  6},      // a[szeregi][kolumny]  
                  { 37, 32, 39, 34, 36},  
                  {537, 532, 539, 534, 536}  
                };  
    int i, j;  
    for (i = 0; i < 3; i++)  
    {   printf("\n Adres początku %d-go szeregu: %x %x\n", i + 1, a[i], *(a + i));  
        for (j = 0; j < 5; j++)  
            printf("Wartość a[%d][%d] elementu: %d\n", i, j, (*(a + i) + j));  
    }  
    return 0;  
}
```

Wakacje na Majorce

```
wakacje (Majorka.La Palma.Petit hotel); wakacje (* ( * (& biuro_turystyczne) ) );  
<----->
```

Znalazłem na mapie.
Wiem, gdzie to jest.
Jakoś tam dotrzemy.



<-----> Znajdę adres biura turystycznego
<-----> Spośród wielu ofert, wybiorę Majorkę
<-----> Wybieram miejsce zakwaterowania
Nieważne są nazwy - organizatorzy
biura zadają o wszystko.



KONIEC POINTERÓW

Łańcuchy znakowe (*Strings*)

String to właściwie jednowymiarowa tablica (*array*) zakończona znakiem **null** (`'\0'`).

```
char pozdrowienie[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

Jest tym samym co:

```
char pozdrowienie[] = "Hello";
```

W czasie inicjalizacji `'\0'` jest wstawiane automatycznie na końcu łańcucha znakowego.

```
// łańcuchy znakowe (strings)
#include <stdio.h>

int main ()
{
    char pozdrowienie[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
    printf("Pozdrowienie: %s\n", pozdrowienie);
    // Sprawdź także działanie poniższych dwóch linii:
    //     printf("Pozdrowienie: %c\n", pozdrowienie[0]);
    //     printf("Pozdrowienie: %c\n", pozdrowienie[1]);
    //     ...
    return 0;
}
```

Przykłady dostępnych funkcji operujących na łańcuchach znakowych:

```
strcpy(s1,s2)      - kopiuje łańcuch znaków s2 do łańcucha znaków s1
strcat(s1,s2)      - dołącza łańcuch znaków s2 na koniec łańcucha znaków s1
strlen(s1)         - zwraca długość łańcucha znaków s1
. . .
```

(Patrz: "Pliki nagłówkowe", "string.h" w celu uzyskania szczegółowych informacji.)

```
// Operacje na łańcuchach znakowych
#include <stdio.h>
#include <string.h>          /* dla strcpy(), strcat() i len() */

int main ()
{
    char str1[15] = "Artur";
    char str2[15] = " Buczek";
    char str3[15];
    int len ;

    strcpy(str3, str1);      /* kopiuje str1 do str3 */
    printf("strcpy(str3, str1) : %s\n", str3 );

    strcat(str1, str2);      /* dołącza str2 do str1 a więc w wyniku daje str1 */
    printf("strcat(str1, str2) : %s\n", str1 );

    len = strlen(str1);      /* całkowita długość str1 po złączeniu łańcuchów */
    printf("strlen(str1)      : %d\n", len );

    return 0;
}
```


Struktury (*Structures*)

Pod jedną nazwą może być wiele zmiennych.

Tak zdefiniowaną strukturę można powielać pod różnymi nazwami i wypełniać je później osobnymi danymi.

```
// Struktury
#include <stdio.h>
#include <string.h>          /* Dla strcpy() */

struct Miasta                /* Prototyp struktury 'Miasta' */
{ char nazwa[15];            /* Nazwa struktury, tu: 'Miasta' zwana jest tag'iem */
  char lokalizacja[45];
  int populacja;
};

int main( )
{ struct Miasta Miasto_1;    /* Deklaracja 'Miasto_1' typu 'struct Miasta' */
  struct Miasta Miasto_2;    /* Deklaracja 'Miasto_2' typu 'struct Miasta' */

  /* Wprowadzenie danych do 'Miasto_1' */
  strcpy( Miasto_1.nazwa, "Warszawa" );          // Miasto_1.nazwa = "Warszawa"
  strcpy( Miasto_1.lokalizacja, "Województwo mazowieckie; centrum Polski" );
  Miasto_1.populacja = 1863000;

  /* Wprowadzenie danych do 'Miasto_2' */
  strcpy( Miasto_2.nazwa, "Kraków");
  strcpy( Miasto_2.lokalizacja, "Województwo małopolskie; południe Polski");
  Miasto_2.populacja = 800000;

  /* drukuj informacje zawarte w 'Miasto_1' */
  printf( "Miasto nr 1, nazwa      : %s\n", Miasto_1.nazwa);
  printf( "Miasto nr 1, lokalizacja : %s\n", Miasto_1.lokalizacja);
  printf( "Miasto nr 1, populacja   : %d\n\n", Miasto_1.populacja);

  /* drukuj informacje zawarte w 'Miasto_2' */
  printf( "Miasto nr 2, nazwa      : %s\n", Miasto_2.nazwa);
  printf( "Miasto nr 2, lokalizacja : %s\n", Miasto_2.lokalizacja);
  printf( "Miasto nr 2, populacja   : %d\n\n", Miasto_2.populacja);

  return 0;
}
```

Prototyp struktury to deklaracja obiektu i jego budowy, która jest gotowa do użycia. To *blueprint*, schemat, rozplanowanie gotowe do wielokrotnego powielania.

Można by pomyśleć o nim jak o pieczętce trzymanej w ręce, na której są pola:

```
nazwa      _____
lokalizacja _____
populacja  _____
```

struct Miasta Miasto_1 to pierwsze odbicie nazwane tu 'Miasto_1' tej pieczętki na kartce papieru a **struct Miasta Miasto_2** to jej następne odbicie, tu o nazwie 'Miasto_2'.

Funkcje `strcpy()` i operator przypisania (`=`) wypełniają puste pola odbitych pieczętek.

Struktury jako argumenty funkcji

Jeżeli struktura będzie argumentem wywoływanej funkcji to znaczy, że przekazujemy do funkcji wszystkie wartości zawarte w tej strukturze.

Wiadomo, że można zwrócić z wywołanej funkcji z powrotem tylko jedną rzecz ale zwracając jedną całą strukturę zwracamy wartości wszystkich jej zmiennych.

Abyś to zapamiętał opowiem krótką historyjkę:

Spotykam listonosza koło domu i chcę przekazać mu parę listów aby zaniósł je na pocztę.

- "Owszem, mam obowiązek odebrać od klienta jakąś rzecz ale co najwyżej jedną", mówi listonosz.

Biorę reklamówkę, wrzucam do niej listy i mówię: "Oto jedna rzecz".

- - - -

Jeżeli w języku *Python* widzisz poprawną instrukcję:

```
return wynik_1, wynik_2, wynik_3, wynik_4
```

to *Python* w tym miejscu sam, o to nie proszony, pakuje wszystkie te wartości do jednego 'tuple' (nasza reklamówka). Nie daj się oszukać, uważaj na szczegóły!

```
// Struktury jako argumenty funkcji
#include <stdio.h>
#include <string.h>

struct Miasta                                /* Prototyp struktury 'Miasta' */
{
    char nazwa[15];
    char lokalizacja[45];
    int populacja;
};

void Drukuj_Miasta_Polskie( struct Miasta miasto );    /* deklaracja funkcji */

int main( )
{
    struct Miasta Miasto_1;                    /* Deklaracja 'Miasto_1' typu 'struct Miasta' */
    struct Miasta Miasto_2;                    /* Deklaracja 'Miasto_2' typu 'struct Miasta' */

    /* Wprowadzenie danych do 'Miasto_1' */
    strcpy( Miasto_1.nazwa, "Warszawa" );
    strcpy( Miasto_1.lokalizacja, "Województwo mazowieckie; centrum Polski" );
    Miasto_1.populacja = 1863000;

    /* Wprowadzenie danych do 'Miasto_2' */
    strcpy( Miasto_2.nazwa, "Kraków");
    strcpy( Miasto_2.lokalizacja, "Województwo małopolskie; południe Polski");
    Miasto_2.populacja = 800000;
    Drukuj_Miasta_Polskie( Miasto_1 );    /* drukuj informacje zawarte w 'Miasto_1' */
    Drukuj_Miasta_Polskie( Miasto_2 );    /* drukuj informacje zawarte w 'Miasto_2' */

    return 0;
}
```

```

void Drukuj_Miasta_Polskie( struct Miasta miasto )
{
    printf( "Nazwa miasta : %s\n", miasto.nazwa);
    printf( "Lokalizacja  : %s\n", miasto.lokalizacja);
    printf( "Populacja    : %d\n\n", miasto.populacja);
}

```

Pointowanie do struktury

```

// Odczyt elementów struktury poprzez pointer
#include <stdio.h>
#include <string.h>

// Zdefiniujmy pointer o typie 'struct Miasta' i nazwie struct_ptr:
//      struct Miasta = *struct_ptr;
// Aby znaleźć adres zmiennej typu struct piszemy & przed nazwą struktury jak
// w przykładzie:
//      struct_ptr = & Miasto_1;
// Przy okazji nie zapominamy, że Miasto_1 jest tym samym co *(& Miasto_1)
// Żeby uzyskać wartość pojedynczej zmiennej (np. nazwa) tej struktury, używamy
// pointera do tej struktury z dwuznakowym zapisem ->
//      struct_ptr -> nazwa;

struct Miasta                                /* definicja struktury */
{
    char nazwa[15];
    char lokalizacja[45];
    int populacja;
};

void Drukuj_Miasta_Polskie( struct Miasta *miasto );      /* prototyp funkcji */

int main( )
{
    struct Miasta Miasto_1;          /* Deklaracja 'Miasto_1' typu 'struct Miasta' */
    struct Miasta Miasto_2;          /* Deklaracja 'Miasto_2' typu 'struct Miasta' */

    /* Wprowadzenie danych do 'Miasto_1' */
    strcpy( Miasto_1.nazwa, "Warszawa" );
    strcpy( Miasto_1.lokalizacja, "Województwo mazowieckie; centrum Polski" );
    Miasto_1.populacja = 1863000;

    /* Wprowadzenie danych do 'Miasto_2' */
    strcpy( Miasto_2.nazwa, "Kraków");
    strcpy( Miasto_2.lokalizacja, "Województwo małopolskie; południe Polski");
    Miasto_2.populacja = 800000;

    /* drukuj zawartość Miasto_1 poprzez przekazanie adresu do Miasto_1 */
    Drukuj_Miasta_Polskie( &Miasto_1 );

    /* drukuj zawartość Miasto_2 poprzez przekazanie adresu do Miasto_2 */
    Drukuj_Miasta_Polskie( &Miasto_2 );

    return 0;
}

```

```

}

// Abyś zapamiętał zapis tłumacz sobie to tak: *miasto => *(&Miasto_1) a to wartość
void Drukuj_Miasta_Polskie( struct Miasta *miasto )
{ printf( "Nazwa miasta : %s\n", miasto->nazwa);
  printf( "Lokalizacja  : %s\n", miasto->lokalizacja);
  printf( "Populacja    : %d\n\n", miasto->populacja);
}

```

Unions

Najpierw 'struktura':

```

// Pamięć zajmowana przez strukturę (dla porównania z 'union')
#include <stdio.h>

struct Zmienne          /* 32 bajty */
{ char c;               /* 1 bajt   */
  int i;                /* 4 bajty */
  float f;              /* 4 bajty */
  double d;            /* 8 bajtów */
  char str[5];         /* 5 bajtów */
};

int main( )
{ struct Zmienne dane;

  printf( "Pamięć zajęta przez dane   : %d\n", sizeof(dane)); // sizeof(Zmienne)
  printf( "Pamięć zajęta przez char   : %d\n", sizeof(dane.c));
  printf( "Pamięć zajęta przez int    : %d\n", sizeof(dane.i));
  printf( "Pamięć zajęta przez float  : %d\n", sizeof(dane.f));
  printf( "Pamięć zajęta przez double : %d\n", sizeof(dane.d));
  printf( "Pamięć zajęta przez ciąg   : %d\n", sizeof(dane.str));
  return 0;
}

```

Jedna taka struktura pochłania 32 bajty ("Pamięć zajęta przez dane").

Zróbmy z niej **union** :

```

// Pamięć zajmowana przez 'union'
#include <stdio.h>

union Zmienne           /* 32 bajty */
{ char c;               /* 1 bajt   */
  int i;                /* 4 bajty */
  float f;              /* 4 bajty */
  double d;            /* 8 bajtów */
  char str[5];         /* 5 bajtów */
};

int main( )
{ union Zmienne dane;

```

```
printf( "Pamięć zajęta przez dane: %d\n", sizeof(dane));    // sizeof(Zmienne)
return 0;
}
```

Wynik:

Pamięć zajęta przez dane: 8

union jest jak struktura, pozwala na trzymanie różnych typów zmiennych z tym, że w określonym czasie trzyma wartość tylko jednej zmiennej - ze wszystkich zmiennych różnych typów - w określonym miejscu jednego zwartego ciągu komórek pamięci, o rozmiarze najbardziej 'bajtożernej' zmiennej.

Aby otrzymać wartość określonej zmiennej, trzeba wcześniej ją 'aktywować'.

Ta idea jest znana z powszechnie stosowanych rekordów (czyli pojedynczych linii) zmiennej długości (np. w technologii 'IBM mainframe'): Przeczytany rekord jest wprowadzany do operacyjnego pola, którego długość deklarujemy nie mniejszą niż długość najdłuższego rekordu z możliwych w czytany pliku, z którego czytamy rekord po rekordzie (nie wszystko na raz). Tam można już bezpiecznie (bo bez jego obciążenia) wprowadzić dowolny pojedynczy rekord w celu jego przetwarzania.

Dostęp do pojedynczej zmiennej union'u

Stosujemy operator dostępu do zmiennej (.) czyli kropkę.

```
// Operator dostępu do elementów union'u i rezultat odczytu danych
#include <stdio.h>

union Zmienne
{
    char znak;
    int liczba;
    double wielka_liczba;
};

int main( )
{
    union Zmienne dane;

    dane.znak = 'Z';
    dane.liczba = 55;
    dane.wielka_liczba = 111111222222.33333;

    printf( "Spodziewane Z a jest: %c\n", dane.znak);
    printf( "Spodziewane 55 a jest: %d\n", dane.liczba);
    printf( "Spodziewane 111111222222.33333 a jest: %lf\n", dane.wielka_liczba);

    return 0;
}
```

Wynik:

Spodziewane Z a jest: U
Spodziewane 55 a jest: -1278323371
Spodziewane 111111222222.33333 a jest: 111111222222.333330

Ostatnio 'dotkniętą' zmienną była 'wielka_liczba' i tylko ona dała poprawny wynik.

Jak chcemy uzyskać wartości wszystkich zmiennych to musimy kolejno je 'aktywować' (wtedy inne zmienne są 'nieaktywne'). Mówiąc niedokładnie ale obrazowo: To jak pointer 'this' w C++. To ten, o który się ostatnio 'zahacza'.

```
// Poprawny odczyt wszystkich elementów union'u
#include <stdio.h>

union Zmienne
{
    char znak;
    int liczba;
    double wielka_liczba;
};

int main( )
{
    union Zmienne dane;

    dane.znak = 'Z';
    printf( "Spodziewane Z a jest: %c\n", dane.znak);

    dane.liczba = 55;
    printf( "Spodziewane 55 a jest: %d\n", dane.liczba);

    dane.wielka_liczba = 1111122222.33333;
    printf( "Spodziewane 1111122222.33333 a jest: %lf\n", dane.wielka_liczba);

    return 0;
}
```

Wynik:

```
Spodziewane Z a jest: Z
Spodziewane 55 a jest: 55
Spodziewane 1111122222.33333 a jest: 1111122222.333330
```

Typ wyliczeniowy (enum)

To zagadnienie dotyczy przypisania nazwom elementów jakiś numerów. Dzięki numerom można dostać ich nazwy; podając nazwy, otrzymujemy ich numery.

Jeżeli mamy małą ilość elementów - powiedzmy do trzech - można im przypisać numery przez dyrektywę (instrukcję dla procesora zaczynającą się znakiem #) #define, np.

```
#define EXIT_SUCCESS 0          ← przykład skopiowany z stdlib.h
#define EXIT_FAILURE 1
```

Jeżeli mamy dużą ilość elementów - powiedzmy powyżej 30 - pozostaje nam odwoływanie się do nich przez ich wartość.

A co można zrobić w przypadku pośrednim? Właśnie zastosować 'typ wyliczeniowy'.

Słowo 'wyliczeniowy' (enumerated) oznacza, że wszystkie elementy tego typu tworzą listę. Jeżeli niewielka ilość elementów da się zgrupować pod wspólną cechą, np. *kolory* to zamiast zapisu:

```
char kolory[] = {"czerwony", "niebieski", "zielony", "czarny"};
```

warto rozważyć zapis:

```
enum kolory {czerwony, niebieski, zielony, czarny};
```

w którym automatycznie **kolejne elementy uzyskują numery: 0, 1, 2, 3.**

W dodatku zapis:

```
enum kolory {czerwony, niebieski, zielony, czarny} kolor;
```

jest równoważny zapisowi dwóch linii:

```
enum kolory {czerwony, niebieski, zielony, czarny};
```

```
enum kolory kolor;
```

i można odwoływać się do poszczególnych elementów tej listy poprzez ich numery.

A jak nam nie odpowiada numeracja od zera, to można zadeklarować np. tak:

```
enum kolory {czerwony = 1, niebieski, zielony, czarny} kolor;
```

```
// Tryb wyliczeniowy (enumeration)
#include <stdio.h>

int main()
{
    enum kolory {czerwony = 1, niebieski, zielony, czarny} kolor;
    for (int i = czerwony; i <= czarny; i++)
        printf("Kolor %d\n", i);
    return 0;
}
```

Wynik:

```
Kolor 1
Kolor 2
Kolor 3
Kolor 4
```

Kolory - zwykle można deklarować je liczbą lub nazwą (wielkie litery):

```
enum COLORS {
    BLACK,          /* dark colors */           // nr 0, 0, czarny
    BLUE,            // nr 1, 1, niebieski
    GREEN,           // nr 2, 2, zielony
    CYAN,            // nr 3, 3, cyjanowy
    RED,             // nr 4, 4, czerwony
    MAGENTA,         // nr 5, 5, purpurowy
    BROWN,          // nr 6, 6, brązowy
    LIGHTGRAY,       // nr 7, 7, jasnoszary
    DARKGRAY,        /* light colors */         // nr 8, 8, ciemnoszary
    LIGHTBLUE,       // nr 9, 9, jasnoniebieski
    LIGHTGREEN,      // nr 10, A, jasnozielony
    LIGHTCYAN,       // nr 11, B, jasny cyjan
    LIGHTRED,        // nr 12, C, jasnoczerwony
    LIGHTMAGENTA,    // nr 13, D, jasnopurpurowy
    YELLOW,          // nr 14, E, żółty
    WHITE            // nr 15, F, biały
};
```

```
#include <stdio.h>
#include <stdlib.h> // dla kolorów w Dev C++
```

```
int main()
{ // "color tło tekst" - poniżej: B - jasnocyjanowe tło, 5 - purpurowy tekst
  system("Color B5");
  printf("\n\n\t\t Przykład z wykorzystaniem 'enum' dla kolorow \n\n");
  return 0;
}
```

Wynik:

Przykład z wykorzystaniem 'enum' dla kolorow

```
-----
Process exited after 1.142 seconds with return value 0
Press any key to continue . . . _
```

typedef - wymiana nazwy typu

Określonej zmiennej (ogólnie: obiektowi) możemy nadać nowy typ.

Poniżej: 'a' to krótka nazwa (alias) typu zastępująca długą: unsigned long long int

```
// Wprowadzenie nowego typu dla zmiennej - typedef (definicja 'typu')
#include<stdio.h>

typedef unsigned long long int a;

int main()
{ a liczba = 1234567890;          /* zmienna 'liczba' jest typu 'a' */
  printf("Duża liczba: %llu\n", liczba);

  return 0;
}
```

Poniżej typ zmiennej **miasto** staje się strukturą **Miasta**. **miasto** jest aliasem **Miasta**. ('alias' znaczy, że nazwy mogą być używane wymiennie a rezultat będzie identyczny.)

```
#include <stdio.h>
#include <string.h>

typedef struct Miasta /* Struktura teraz może nawet nie mieć tagu: struct {...} */
{ char nazwa[15];      /* Tak więc słowo 'Miasta' można stąd usunąć bo */
  char lokalizacja[45]; /* poniżej nigdy się na niego nie powołujemy */
  int populacja;
} miasto;             /* 'miasto' jest teraz tym samym co 'struct Miasta {...}' */

int main( )
{ miasto Miasto;      /* Zmienna 'Miasto' o typie 'miasto' */

  /* Wprowadzenie danych do 'Miasto' */
  strcpy( Miasto.nazwa, "Warszawa" );
  strcpy( Miasto.lokalizacja, "Województwo mazowieckie; centrum Polski" );
  Miasto.populacja = 1863000;
```



```

/* drukuj informacje zawarte w 'Miasto' */
printf( "Nazwa miasta : %s\n", Miasto.nazwa);
printf( "Lokalizacja  : %s\n", Miasto.lokalizacja);
printf( "Populacja    : %d\n", Miasto.populacja);

return 0;
}

```

typedef w porównaniu z #define

#define jest dyrektywą języka C, która jest również używana do definiowania aliasów dla różnych typów danych podobnie do **typedef**, ale z następującymi różnicami:

- **typedef** ogranicza się do nadawania symbolicznych nazw typom tylko wtedy, gdy jako **#define** może być użyty do zdefiniowania aliasu dla wartości, np. można zdefiniować 1 jako JEDEN albo PRAWDA itp.
- Instrukcja **typedef** jest przerabiana przez kompilator, podczas gdy instrukcja **#define** jest przetwarzana przez preprocesor - dla kompilatora to 'niekwestionowane założenie'.

```

#include<stdio.h>

#define SSS 1/137                // stała struktury subtelnej

int main()
{ printf("To ja, najbardziej tajemnicza liczba Natury: %.7f...\n\n", (float)SSS );
  return 0;
}

```

Pliki I/O

Otwieranie pliku

Aby cokolwiek robić z jakimś plikiem, musisz go najpierw otworzyć.

Aby otworzyć plik (istniejący lub nowy) używamy funkcji **fopen()** typu **FILE**.

```
FILE *fopen( "ścieżka dostępu do pliku", "tryb" )
```

Przykłady:

```
FILE *fp = fopen("C:/Users/Artur/Desktop/test.txt", "r");           // tylko do odczytu
lub
```

```
FILE *fp;
fp = fopen("C:/Users/Artur/Desktop/test.txt", "w");                // tylko do wpisywania do pliku
```

tryb	opis działania trybu
r	Otwiera istniejący plik tekstowy dla jego odczytu.
w	Otwiera plik tekstowy do wpisywania a istniejący tekst ginie. Jeżeli plik nie istnieje, nowy plik jest tworzony.
a	Otwiera plik tekstowy do dopisywania a więc stary tekst nie ginie. Jeżeli plik nie istnieje, nowy plik jest tworzony.
r+	Otwiera plik tekstowy zarówno do odczytu jak i wpisywania. Nie niszczy starego tekstu.
w+	Otwiera plik tekstowy zarówno do odczytu jak i wpisywania. Niszczy stary tekst. Jeżeli plik nie istnieje, nowy plik jest tworzony.
a+	Otwiera plik tekstowy zarówno do odczytu jak i wpisywania. Dopisuje do istniejącego tekstu a więc on nie ginie. Jeżeli plik nie istnieje, nowy plik jest tworzony.

Jeżeli używamy pliku binarnego to do symbolu trybu trzeba dopisać **b**
"rb", "wb", "ab", "rb+", "r+b", "wb+", "w+b", "ab+", "a+b"

Zamykanie pliku

```
fclose( FILE *fp ); // czyli: int fclose( pointer do pliku ); <-- z stdio.h
```

gdzie **fp** mogłoby być zdefiniowane tak:

```
fp = fopen("c:/Users/ Twój identyfikator /desktop/test.txt", "w+");
```

```
np. fp = fopen("c:/Users/Artur/desktop/test.txt", "w+");
```

albo, jeżeli środowisko pracy języka C jest już w katalogu "c:/Users/Artur":

```
fp = fopen("/desktop/test.txt", "w+");
```

Wpisywanie do pliku

```
fprintf( pointer do pliku, tekst )
```

Warto postępować zgodnie z ustalonymi zasadami:

Jak piszesz `printf(...)` to wynik wypisuje na ekran.

Jak piszesz `fprintf(...)` to wynik wpisuje do pliku.

Jak piszesz `cprintf(...)` to sformatowany (np. kolorem) wynik wypisuje na ekran.

Jak piszesz `sprintf(...)` to wynik wpisuje do określonego bufora.

```
fwrite( tekst, rozmiar bloku tekstu, ilość 'rozmiaru bloku tekstu', pointer do pliku )
```

Funkcja ***fwrite()*** wpisuje łańcuch znakowy (tu 'tekst') do pliku określonego pointerem.

Jeżeli 'rozmiar bloku tekstu' jest całym tekstem, to ilość 'rozmiaru bloku tekstu' = 1.

```
fputs( tekst, pointer do pliku );
```

Funkcja ***fputs()*** wpisuje łańcuch znakowy (tu 'tekst') do pliku określonego pointerem.

```
fputc( znak, pointer do pliku );
```

Funkcja ***fputc()*** wpisuje znak do pliku określonego pointerem.

```
// Test na działanie funkcji wpisywania do pliku: fprintf(), fwrite(), fputs() i fputc()
```

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

```
#include <string.h> // dla strlen()
```

```
int main()
```

```
{ char tekst_1[] = "To jest test na działanie 'fprintf()'...";
```

```
char tekst_2[] = "To jest test na działanie 'fwrite()'...";
```

```
char tekst_3[] = "To jest test na działanie 'fputs()'...";
```

```
char tekst_4[] = "To jest test na działanie 'fputc()'...";
```

```
FILE *fp; // fp sugeruje nazwę 'file pointer'
```

```
fp = fopen("C:/Users/Artur/Desktop/test.txt", "w+"); // daj swój identyfikator
```

```
fprintf(fp, tekst_1); fprintf(fp, "\n");
```

```
fwrite(tekst_2, sizeof(tekst_2), 1, fp ); fprintf(fp, "\n");
```

```
fputs(tekst_3, fp); fprintf(fp, "\n");
```

```
for(int i=0; i<strlen(tekst_4); i++)
```

```
fputc(tekst_4[i], fp);
```

```
fclose(fp);
```

```
return 0;
```

```
}
```

Wynik: Na panelu tworzy plik "test.txt" a w nim następujący tekst:

```
To jest test na działanie 'fprintf()'...
To jest test na działanie 'fwrite()'...
To jest test na działanie 'fputs()'...
To jest test na działanie 'putc()'...
```

Zdania kończą się znakiem [Enter], który da o sobie znać przy czytaniu pliku.

Czytanie pliku

Zwykle dane czytane są do wyznaczonego przez nas miejsca a będzie to zmienna przyjmująca łańcuch znakowy i nazwiemy ją tu buforem [dołączając na końcu znak null ('\0') kończący zapis.] i dopiero stąd dane te wypisywane są np. na ekran.

```
fscanf( pointer do pliku, "%format", bufor )
```

Czytanie przez **fscanf()** zakończy pierwsza spacja.

Tak jak scanf(), także fscanf() może czytać wiele wartości jednocześnie.

```
fread( bufor, rozmiar bloku, ilość 'rozmiaru bloku', pointer do pliku )
```

'rozmiar bloku' to zwykle rozmiar bufora a wtedy ilość 'rozmiaru bloku' = 1.

```
fgets( bufor, n, pointer do pliku );
```

Funkcja **fgets()** czyta do n-1 znaków z pliku określonego przez pointer. Kopiuje przeczytany tekst do bufora. Znak [Enter] przerywa czytanie a więc czyta linię.

```
fgetc( pointer do pliku );
```

Funkcja **fgetc()** czyta pierwszy napotkany znak z pliku określonego przez pointer.

```
// Test na działanie funkcji czytania z pliku: fscanf(), fread(), fgets() i fgetc()
#include <stdio.h>

int main()
{   FILE *fp;
    char bufor[100];

    fp = fopen("C:/Users/Artur/Desktop/test.txt", "r"); // zmień identyfikator na swój

    fscanf(fp, "%s", bufor);
    printf("fscanf() test: %s\n", bufor );

    fread(bufor, 20, 1, fp);                                // czyta maksymalnie 20 znaków
    printf("fread() test : %s\n", bufor );

    fgets(bufor, 100, fp);
    printf("fgets() test : %s\n", bufor );

    printf("fgetc() test : %c\n", fgetc(fp) );

    fclose(fp);
    return 0;
}
```

Wynik:

```
fscanf() test: To          <-- czytanie przerwane spacją
fread() test :  jest test na działa <-- czyta dalej 20 znaków; spacja jest pierwsza
fgets() test :  nie 'fprintf()'...   <-- czytanie przerwane niewidzianym znakiem [Enter]
                                     <-- pusta linia wymuszona przez zakończenie fgets()
fgetc() test :  T               <-- pierwszy znak nowej linii
```

Kursor odczytu z pliku i wpisu do pliku

Wszelki odczyt i zapis związany jest z istnieniem niewidocznego kursora, który ten odczyt i zapis prowadzi. Po otwarciu pliku kursor czytania/pisania jest na jego początku (pozycja 0). Nie dziw się, że po przeczytaniu pliku, nie możesz go ponownie przeczytać bez odnowienia początkowej pozycji kursora.

`fseek(pointer do pliku, ustawienie początku odczytu, sposób ustawienia kursora)`

Sposoby ustawienia kursora:

SEEK_SET - ustaw kursor czytania na początek pliku

SEEK_CUR - ustaw kursor czytania na podane miejsce

SEEK_END - ustaw kursor czytania na koniec pliku

`ftell(pointer do pliku)`

Funkcja **ftell()** podaje aktualną pozycję kursora w pliku od jego pierwszego bajtu.

`rewind(pointer do pliku)`

Funkcja **rewind()** ustawia kursor na początek pliku, podobnie do **SEEK_SET** z ustawieniem 0.

```
// Test na działanie kursora odczytu i wpisu
#include <stdio.h>
#include <stdlib.h>          // dla calloc()

int main ()
{
    FILE *fp;
    char *bufor;
    long rozmiar_pliku;      // 'long' jest tym samym co 'long int'
    char znak;

    fp = fopen ("C:/Users/Artur/Desktop/test.txt","r"); // zmień identyfikator na swój

    fseek(fp, 0, SEEK_END);  // ustawienie kursora na końcu pliku bez jakiegokolwiek
                             // przesunięcia (0) w stosunku do końca pliku
    rozmiar_pliku = ftell(fp); // pozycja ostatniego bajtu to zarazem długość tekstu
    printf("Rozmiar pliku w bajtach: %ld \n\n", rozmiar_pliku);

    // przydzielenie pamięci pointerowi 'bufor'. Tam będzie odczytany z pliku tekst.
    bufor = (char*)calloc(rozmiar_pliku, sizeof(char));

    fseek(fp, 0, SEEK_SET);  // ustawienie kursora z powrotem na początek pliku
    printf("Zawartość pliku: \n");
    do
    {
        znak = fgetc(fp);
        printf("%c", znak);    // wrzuc tekst (znak po znaku) na ekran monitora
    } while (znak != EOF);     // EOF - 'End Of File' zdefiniowane w 'stdio.h':
                                //      #define EOF (-1)

    // ustaw kursor na pozycji 84-ej (pierwsza pozycja to 0) i czytaj do końca
    fseek(fp, 84, SEEK_SET);
    fread ( bufor, sizeof(char), rozmiar_pliku, fp );
```

```

printf("\n\nPo SEEK_SET ustawionym na 84-ym bajcie pliku: \n%s\n", bufor);

// z aktualnej pozycji kursora, cofnij kursor o 52 pozycje
fseek(fp, -52, SEEK_CUR);
fgets ( bufor, rozmiar_pliku, fp );
printf("\nPo SEEK_CUR ustawionym o 52 pozycje wzczesniej do znaku [Enter] - %s", bufor);

// kursor będuacy na koncu pliku, cofnij o 12 pozycji
fseek(fp, -12, SEEK_END);
fgets ( bufor, rozmiar_pliku, fp );
printf("\nPo SEEK_END ustawionym o 12 pozycji wzczesniej niz koniec pliku - %s", bufor);

// wróc z kursorem na początek pliku
rewind(fp); // tak samo dziala jak: fseek(fp, 0, SEEK_SET);

fclose(fp);
return 0;
}

```

Wynik:

Rozmiar pliku w bajtach: 162

Zawartość pliku:

To jest test na dzialanie 'fprintf()'
 To jest test na dzialanie 'fwrite()'
 To jest test na dzialanie 'fputs()'
 To jest test na dzialanie 'fputc()'

Po SEEK_SET ustawionym na 84-ym bajcie pliku:

To jest test na dzialanie 'fputs()'
 To jest test na dzialanie 'fputc()'

Po SEEK_CUR ustawionym o 52 pozycje wzczesniej do znaku [Enter] - 'fputs()'

Po SEEK_END ustawionym o 12 pozycji wzczesniej niz koniec pliku - 'fputc()'

Preprocesor

Preprocesor nie jest częścią kompilatora.

Instrukcje preprocesora czytane są **przed** kompilacją.

#define	Podstawia makra
#include	Wstawia określony plik nagłówkowy (<i>header</i>) z zewnątrz programu
#undef	Oddefiniowuje (definicja przestaje obowiązywać) makra
#ifdef	Zwraca prawdę jeżeli makro jest zdefiniowane
#ifndef	Zwraca prawdę jeżeli makro nie jest zdefiniowane
#if	Testuje czy stan warunku jest prawdą
#else	Alternatywa dla <i>#if</i>
#elif	<i>#else</i> i <i>#if</i> w jednym zapisie
#endif	Kończy warunek preprocesora
#error	Drukuje wiadomość o błędzie na <i>stderr</i> (czyli ekranie monitora)
#pragma	Uruchamia specjalne funkcje dla kompilatora

Przykłady preprocesora

```
#define MAX 20
```

```
#include <stdio.h>
#include "myheader.h"
```

```
#undef FILE_SIZE
#define FILE_SIZE 42
```

```
#ifndef MESSAGE
    #define MESSAGE "Artur Buczek prezentuje"
#endif
```

```
#ifdef DEBUG
    /* Tutaj piszesz instrukcje dotyczące błędu */
#endif
```

Predefiniowane makra

Tylko pięć z nich ze środowiska 'Turbo C++' pracuje w 'Dev C++':

__DATE__	Aktualny czas w formacie znakowym "MMM DD YYYY", np. Nov 11 2024
__TIME__	Aktualny czas w formacie znakowym "HH:MM:SS"
__FILE__	Zawiera aktualną nazwę pliku w zapisie znakowym
__LINE__	Zawiera aktualny numer linii w zapisie numerycznym
__STDC__	Zdefiniowane jako 1 gdy kompilator kompiluje w standardzie ASCII

```
// Wbudowane makra
#include<stdio.h>

int main()
{   printf("Plik   : %s\n", __FILE__ );
    printf("Data   : %s\n", __DATE__ );
    printf("Czas   : %s\n", __TIME__ );
    printf("Linia  : %d\n", __LINE__ );
    printf("Ansi   : %d\n", __STDC__ );
    return 0;
}
```

Warto wiedzieć, który program przetworzył dane jakie będą poniżej i kiedy to zrobił.

```
// Przykład wykorzystania predefiniowanych makr dla udokumentowania otrzymanych danych
#include<stdio.h>

int main()
{   printf("Program %s, z dnia %s, %s\n", __FILE__ , __DATE__ , __TIME__ );
    printf("\n\n    Tu będzie wynik działania programu\n");
    return 0;
}
```

Wynik:

```
Program C:\Users\Artur\Desktop\Program1.cpp, z dnia May 12 2024, 17:39:47
```

Tu będzie wynik działania programu

Gdybyś chciał to zrobić jeszcze lepiej, patrz: "Pliki nagłówkowe", "time.h".
Makra te są używane przez inne makra i funkcje, np. `assert()`, patrz: *Header* "assert.h".

Pliki nagłówkowe (*headers*)

Pliki z rozszerzeniem `.h`, które zawierają już zadeklarowane funkcje języka `C` i makra, mogą być dołączane do wielu osobnych programów.

A są one dołączane do programów dyrektywą `#include` i jest standardem u programistów, że pomiędzy `< i >` to *header*'y zdefiniowane firmowo (przez *Microsoft* lub firmę *Borland*) a pomiędzy `" i "` zdefiniowane przez użytkownika (np. Ciebie).

(Patrz: "Pliki nagłówkowe" w celu uzyskania szczegółowych informacji.)

Header	Opis zawartych funkcji	Przykładowe funkcje
stdio.h	Operacje wejścia i wyjścia	<code>fopen()</code> , <code>fclose()</code> , <code>fread()</code> , <code>fwrite()</code> <code>getc()</code> , <code>getch()</code> i <code>getche()</code> <i><-- z 'conio.h'</i> , <code>putc()</code> , <code>getchar()</code> , <code>putchar()</code> , <code>gets()</code> , <code>puts()</code> , <code>fgetc()</code> , <code>fputc()</code> , <code>fgets()</code> , <code>fputs()</code> , <code>ungetc()</code> <code>scanf()</code> , <code>fscanf()</code> , <code>sscanf()</code> , <code>printf()</code> , <code>fprintf()</code> , <code>sprintf()</code> , <code>snprintf()</code> <code>ftell()</code> , <code>fseek()</code> razem z: <code>SEEK_SET</code> , <code>SEEK_CUR</code> i <code>SEEK_END</code> , <code>rewind()</code> , <code>fgetpos()</code> , <code>fsetpos()</code> <code>clearerr()</code> , <code>feof()</code> , <code>ferror()</code> , <code>perror()</code> <code>remove()</code> , <code>rename()</code> , <code>tmpnam()</code>
string.h	Operacje na łańcuchach znakowych	<code>strlen()</code> , <code>strcat()</code> , <code>strncat()</code> , <code>strcpy()</code> , <code>strncpy()</code> , <code>strcmp()</code> , <code>strncmp()</code> , <code>strcoll()</code> , <code>strchr()</code> , <code>strrchr()</code> , <code>strspn()</code> , <code>strcspn()</code> , <code>strpbrk()</code> , <code>strstr()</code> , <code>strlwr()</code> , <code>strupr()</code> , <code>strrev()</code>
conio.h	Operacje wejścia i wyjścia dla konsoli	<code>system("cls")</code> , <code>kbhit()</code> , <code>getch()</code> , <code>getche()</code> , <code>putch()</code> , <code>ungetch()</code>
stdlib.h	Funkcje biblioteki standardowej: zarządzanie pamięcią, narzędzia programowe, konwersje ciągów znaków, liczby losowe, algorytmy	<code>atof()</code> , <code>atoi()</code> , <code>atol()</code> , <code>itoa()</code> , <code>ltoa()</code> , <code>strtod()</code> , <code>strtol()</code> , <code>strtoul()</code> , <code>strtoll()</code> , <code>strtoull()</code> <code>rand()</code> , <code>srand()</code> <code>malloc()</code> , <code>calloc()</code> , <code>realloc()</code> , <code>free()</code> , <code>calloc()</code> <code>abort()</code> , <code>exit()</code> , <code>system()</code> <code>qsort()</code> , <code>bsearch()</code> <code>abs()</code> , <code>labs()</code> , <code>div()</code> , <code>ldiv()</code>
math.h	Stałe i funkcje matematyczne	Stałe matematyczne: <code>M_E</code> , <code>M_LOG2E</code> , <code>M_LOG10E</code> , <code>M_LN2</code> , <code>M_LN10</code> , <code>M_PI</code> , <code>M_PI_2</code> , <code>M_PI_4</code> , <code>M_1_PI</code> , <code>M_2_PI</code> , <code>M_2_SQRTPI</code> , <code>M_SQRT2</code> , <code>M_SQRT_2</code> Funkcje: <code>round()</code> , <code>floor()</code> , <code>ceil()</code> , <code>trunc()</code> , <code>fmod(y)</code> <code>sin()</code> , <code>cos()</code> , <code>tan()</code> , <code>asin()</code> , <code>acos()</code> , <code>atan()</code> , <code>atan2()</code> <code>sinh()</code> , <code>cosh()</code> , <code>tanh()</code> , <code>asinh()</code> , <code>acosh()</code> , <code>atanh()</code> <code>sqrt()</code> , <code>cbrt()</code> , <code>exp()</code> , <code>exp2()</code> , <code>pow()</code> , <code>hypot()</code> <code>log()</code> , <code>log10()</code> , <code>log2()</code> <code>fdim()</code> , <code>fmin()</code> , <code>fmax()</code> <code>abs()</code> , <code>fabs()</code>
ctype.h	Operacje na typach znakowych	<code>isalnum()</code> , <code>isalpha()</code> , <code>iscntrl()</code> , <code>isdigit()</code> , <code>isxdigit()</code> , <code>isgraph()</code> , <code>islower()</code> , <code>isupper()</code> , <code>isprint()</code> , <code>ispunct()</code> , <code>isspace()</code> , <code>isblank()</code> , <code>tolower()</code> , <code>toupper()</code>
time.h	Makro, struktury, definicje typu i funkcje daty i czasu	<code>CLOCKS_PER_SEC</code> <code>tm{}</code> , <code>size_t</code> , <code>time_t</code> , <code>clock_t</code> , <code>asctime()</code> , <code>ctime()</code> , <code>gmtime()</code> , <code>clock()</code> , <code>difftime()</code> , <code>time()</code> , <code>localtime()</code> ,

		mktime(), strftime()
errno.h	Operacje obsługi błędów	errno(), perror() strerror() <-- nie z 'errno.h' lecz z 'string.h'
locale.h	Funkcje umożliwiające modyfikację programu dla bieżących ustawień regionalnych, na których jest uruchomiony.	setlocale(), localeconv(), lconv()
signal.h	Funkcje obsługujące sygnał	signal(), raise()
stdarg.h	Makra obsługi funkcji o zmiennej liczbie argumentów różnych typów	va_list(), va_start(), va_arg(), va_end()
limits.h	Podaje ograniczenia zmiennych	CHAR_BIT, SCHAR_MIN, SCHAR_MAX, UCHAR_MAX, CHAR_MIN, CHAR_MAX, SHRT_MIN, SHRT_MAX, USHRT_MAX, INT_MIN, INT_MAX, UINT_MAX, LONG_MIN, LONG_MAX, ULONG_MAX, LLONG_MIN, LLONG_MAX, ULLONG_MAX, MB_LEN_MAX
assert.h	Zawiera informacje dotyczące dodawania diagnostyki, które ułatwiają debugowanie programu.	assert()

Co to naprawdę jest dyrektywa `#include<plik_języka_C>` lub `#include"plik_języka_C"`, jaka jest jej funkcja?

- Zasadniczą funkcją jest dołączanie istniejących 'fabrycznie' (np. od Borlanda) plików nagłówkowych,
- Dołączanie własnych plików nagłówkowych wzbogacających funkcjonalność oprogramowania w ściśle określonym celu, np. wysoko wydajna statystyka matematyczna.

Przed wszystkim należy pamiętać, że dyrektywa `#include` wkopiuje w to miejsce wymagany plik. Oto dowód:

Na pulpicie (desktop) utworzyłem w Notatniku swój 'plik nagłówkowy' (*header*) o nazwie ToMoje, z rozszerzeniem .h:

```
for( int i = 0; i < 10; i++ )
    printf(" %d", i );
return 0;
```

Zapisując go jako (Zapisz jako...) Nazwa pliku: ToMoje.h Zapisz jako typ: Wszystkie pliki
Kodowanie: UTF-8 i uruchomiłem program:

```
// Włączanie zewnętrznego kodu do wnętrza programu
#include<stdio.h>

int main()
{
    #include"C:\Users\Artur\desktop\ToMoje.h"
}
```

Wynik:

```
0 1 2 3 4 5 6 7 8 9
```

Taki sam efekt można uzyskać, gdy kod z notatnika zapiszemy jako ToMoje.cpp i w kodzie programu linię z 'include' zamienimy na `#include"C:\Users\Artur\desktop\ToMoje.cpp"` .

Casting (rzut)

Umożliwia wymianę typu zmiennej w trakcie działania programu.

Np. $11/8$ daje w wyniku 1 bo operacje na `int` dają w wyniku też `int`. *Casting* (rzut) umożliwia odczyt wyniku jako `float` lub `double`.

```
// Rzut (casting)
#include <stdio.h>

int main()
{   int dzielna = 11, dzielnik = 8;
    float iloraz;

    iloraz = dzielna / dzielnik;
    printf("Wynik dzielenia (niepoprawny) : %f\n", iloraz );

    iloraz = (float) dzielna / dzielnik;          // <-- przykład 'rzutu' int na float
    printf("Wynik dzielenia (poprawny)      : %f\n", iloraz );
    return 0;
}
```

Zwróć uwagę na zapis 'rzutu' odróżniającego go od zapisu 'funkcji', mogącej robić podobne rzeczy.

```
#include<stdio.h>
#include<stdlib.h>    // dla atoi, atof, ...

int main()
{   int n;
    char str[] = "12345";

    n = atoi(str);    // zamiana łańcucha znakowego na liczbę (o ile da się to zrobić)
    printf("\n znaki = %s \n liczba = %d \n liczba + 5 = %d", str, n, n + 5);
    return 0;
}
```

Obsługa błędów (*Error handling*)

Kod powrotu 0 oznacza brak błędu składni języka C. Nie oznacza to braku błędu logiki programu. Funkcje `perror()` i `strerror()` wyświetlają komunikat z `errno` związany z napotkanym błędem.

(Patrz: "Pliki nagłówkowe", "errno.h" w celu uzyskania szczegółowych informacji.)

```
#include <stdio.h>
#include <errno.h>
#include <string.h>          // dla strerror()

extern int errno ;          // nie musimy tego tu deklarować bo to już jest w 'errno.h'

int main ()
{   FILE *fp;
    fp = fopen ("Nie_ma_mnie.txt", "r");
```

```

if (fp == NULL)
{
    fprintf(stderr, "Numer błędu: %d\n", errno);
    perror("Błąd drukowany przez 'perror' ");
    fprintf(stderr, "Błąd otwierania pliku: %s\n", strerror( errno ));
} else
    fclose (fp);

return 0;
}

```

Wynik:

```

Numer błędu: 2
Błąd drukowany przez perror: No such file or directory
Błąd otwierania pliku: No such file or directory

```

Zarządzanie pamięcią (*Memory management*)

Tych kilka funkcji dynamicznej* alokacji pamięci** i uwolnienia tej pamięci zawartych jest 'Dev C++' w <stdlib.h>

- - - - -

* Słowo 'dynamicznej' znaczy 'w trakcie działania programu' (*runtime*).

** Wyrażenie 'alokacja pamięci' znaczy 'umieszczenie określonej ilości bajtów'.

void *calloc(*liczba_bloków*, int *liczba_bajtów*) - (contiguous **allocation**) alokuje określoną liczbę (*liczba_bloków*) bloków, każdy o rozmiarze *liczba_bajtów* i inicjalizuje ją zerami

void *malloc(int *liczba_bajtów*) - (memory **allocation**) alokuje pojedynczy blok pamięci z określoną liczbą bajtów

void *realloc(void **adres*, int *nowy_rozmiar*) - zmienia alokację pamięci daną wcześniej przez *calloc()* lub *malloc()* gdy zaalokowana pamięć okazuje się za mała

void free(void **adres*) - uwalnia pamięć w bloku wyszczególnionym w adresie *adres*
(Patrz: "Pliki nagłówkowe", "stdlib.h" w celu uzyskania szczegółowych informacji.)

Dynamiczna alokacja pamięci i jej uwolnienie

```

// Alokowanie (rezerwacja) i uwalnianie określonej ilości komórek pamięci
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

int main()
{
    char *opis;

    opis = (char *) malloc( 50 * sizeof(char) );    /* dynamiczna alokacja pamięci */

    if ( opis == NULL )

```

```

    fprintf(stderr, "Error - nie można zaalokować wymaganej ilości pamięci \n",
strerror( errno ) );
    else
        strcpy( opis, "Jestem studentem Akademii Frycza w Krakowie");

    printf("Opis: %s\n", opis );

    /* uwolnienie zaalokowanej pamięci */
    free(opis);
    return 0;
}

```

Dynamiczna zmiana rozmiaru pamięci i jej uwolnienie

```

// Zmiana rozmiaru wcześniej alokowanej pamięci i jej całkowite uwolnienie
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

int main()
{
    char *opis;

    opis = (char *) malloc( 50 * sizeof(char) );    /* dynamiczna alokacja pamięci */

    if ( opis == NULL )
        fprintf(stderr, "Error - nie można zaalokować wymaganej ilości pamięci \n",
strerror( errno ));
    else
        strcpy( opis, "Jestem studentem Akademii Frycza w Krakowie");

    printf("Opis: %s\n\n", opis );

    /* założmy, że chcemy mieć dłuższy opis - już 50 bajtów nie wystarczy */
    opis = (char *) realloc( opis, 90 * sizeof(char) );

    if ( opis == NULL )
        fprintf(stderr, "Error - nie można zaalokować wymaganej ilości pamięci \n",
strerror( errno ));
    else
        strcat( opis, " i systematycznie robię postępy w nauce.");

    printf("Opis: %s\n\n", opis );

    /* uwolnienie całej zaalokowanej pamięci */
    free(opis);

    return 0;
}

```