

NAZWA PRZEDMIOTU
Programowanie strukturalne

Temat wykładu 11.

**Złożone typy danych: struktury, unie, pola bitowe, wyliczenie enum,
definicje nazwy typu typedef.**

dr hab. inż. Jerzy Montusiewicz, prof. PL

11. Agenda

11.1. Złożone typy danych.

11.2. Struktury.

11.3. Unie.

11.4. Pola bitowe.

11.5. Wyliczenie enum.

11.6. Definicje nazwy typu typedef.

11.1. Złożone typy danych

Złożone typy danych pozwalają, w przypadku **typu strukturalnego**, na łatwiejszy sposób przechowywania wielu danych różnego typu, które odnoszą się do pojedynczego obiektu.

Pozwalają, w przypadku **unii**, na przekazywanie informacji między różnymi zmiennymi przez korzystanie z tego samego obszaru pamięci.

Pola bitowe umożliwiają zdefiniować zestaw danych, w których każdy z elementów ma rozmiar określony w pojedynczych bitach.

Typ wyliczeniowy **enum** pozwala na tworzenie zmiennych, które przyjmują wartości stałych całkowitych.

Typedef służy do definiowania nowych nazw (identyfikatorów) istniejącym już typom.

11.2. Struktury, **struct**

Struktura to złożony typ danych definiowany przez programistę, która umożliwia przechowywanie w pojedynczej zmiennej wiele wartości różnego typu. Tak więc struktura zawiera pola będące zestawem heterogenicznych danych (np. int, float, char, tablic różnego typu).

Możliwe jest wykonywanie pewnych operacji na zestawie (strukturze) jako całości oraz na wybranych elementach struktury.

Definiowanie struktury jest dwuetapowe:

1 ETAP: definiowanie nazwy i pól

```
struct adres {
```

Nazwa struktury

```
    char nazwisko[20];
```

Pola struktury różnego typu

```
    char *miejsce;
```

To jest tylko definicja typu,

```
    long ps; };
```

nie istnieją żadne zmienne tego typu.

2 ETAP: powołanie konkretnego egzemplarza/y typu strukturalnego

```
struct adres osoba1, osoba2;
```

11.2. Struktury, **struct**

Powołano dwie struktury **osoba1** i **osoba2** typu strukturalnego **adres**.

Połączenie obu etapów definiowania:

```
struct adres {  
    char nazwisko[20];  
    char *miejsce;  
    long ps;  
} osoba1, osoba2;
```

Można tworzyć wiele struktur gdy podajemy ich nazwy. Gdy zrezygnujemy z nazwy w pierwszej definicji wtedy można zdefiniować tylko jedną strukturę (anonimową).

```
struct {  
    int nazwa1;  
    float nazwa2; } x, y, z;
```

11.2. Struktury, **struct**

Stosowanie struktury:

- w takim zasięgu gdzie była umieszczona:
- w zasięgu globalnym (w całym programie),
- w funkcji.

Dostęp do elementów składowych obiektów struktury:

- **operator kropki** – łączący nazwę zmiennej strukturalnej z nazwą elementu składowego
`osoba1.ps = 121;`
`gets (osoba1.nazwisko);`
- **operator strzałki** – łączy obiekt reprezentowany przez nazwę wskaźnika
`gets (osoba1->miejsce);`

Postać `osoba1->miejsce` jest w zasadzie skrótem zapisu `(*osoba1).miejsce`, gdyż `*osoba1` jest wartością obiektu wskazywanego przez `osoba1`.

*Równoważność
zapisów.*

<code>e.v = 2.71;</code>	\Leftrightarrow	<code>(&e)->v = 2.71;</code>
<code>pe->v = 2.71;</code>	\Leftrightarrow	<code>(*pe).v = 2.71;</code>

11.2. Struktury, **struct**, p11-2a

Dostęp do obiektów i do całości struktury.

```
int main (void)
```

```
{  
    struct adres {  
        char nazwisko[20];  
        char miejsce[30];  
        long ps;  
    };  
    struct adres osoba1, *osoba2, *osoba3;  
    printf("Wprowadz nazwisko\t"); gets(osoba1.nazwisko);  
    printf("Wprowadz miejsce\t"); gets(osoba1.miejsce);  
    printf("Wprowadz date rrrrmmdd\t"); scanf("%d",&osoba1.ps);  
    printf ("\nZawartosc struktury osoba1:\n");  
    printf ("%d  %s  %s\n", osoba1.ps, osoba1.nazwisko, osoba1.miejsce);  
    osoba2=&osoba1;  
    printf ("\nZawartosc struktury osoba2:\n");  
    printf ("%d  %s  %s\n", osoba2->ps, osoba2->nazwisko, osoba2->miejsce);  
    osoba3=osoba2;  
    printf ("\nZawartosc struktury osoba3:\n");  
    printf ("%d  %s  %s\n", osoba3->ps, osoba3->nazwisko, osoba3->miejsce);  
    return 0;  
}
```

Definiowanie struktury 'adres' – 1 etap.

Etap 2 – powołanie egzemplarzy (obiektów).

Wprowadzenie wartości do obiektu struktury.

Kopiowanie struktury

Kopiowanie struktury

11.2. Struktury, **tablica struktur**, p11-2b

Dostęp do elementów.

```
int main (void)
{struct sample
{ int i;
  double d;
  char *napis;
} one[3];
printf("Wprowadz int\t");
scanf("%d",&one[0].i);
one[0].d = 98.6;
strcpy (one[0].napis, "Politechnika");
printf ("Zawartosc struktury one[0]:\n");
printf ("%d\t %5.2lf\t %s\n", one[0].i, one[0].d, one[0].napis);
one[2]=one[0];
printf (" ++Zawartosc struktury one[2]:\n");
printf ("%d\t %5.2lf\t %s\n", one[2].i, \
one[2].d, one[2].napis);
return 0;
}
```

Definiowanie struktury
'sample' – jednoetapowe.

Powołanie egzemplarzy
(obiektów) jako tablicy.

Wprowadzenie
wartości do
obiektu struktury.

Kopiowanie struktury

Do przykładu p11.2a

```
Wprowadz nazwisko      Kowalski
Wprowadz miejsce       Lublin
Wprowadz date rrrrmmdd 19990710
```

```
Zawartosc struktury osoba1:
19990710      Kowalski      Lublin
```

```
Zawartosc struktury osoba2:
19990710      Kowalski      Lublin
```

```
Zawartosc struktury osoba3:
19990710      Kowalski      Lublin
```

```
Wprowadz int      123
Zawartosc struktury one[0]:
123      98.60      Politechnika
++Zawartosc struktury one[2]:
123      98.60      Politechnika
```


11.2. Struktury, **struktura globalna**, p11-2c

```
struct Narty {  
    double cena;  
    int rok;  
    char model[10]; // nie mogę *model  
} head = {.cena = 1350}, fischer = {1299, 2018, "slalom"};  
  
int main() {  
    struct Narty blizard, *mojenar = &fischer;  
    printf("Rozmiar obiektow struktuty 'Narty'=%d bajty\n ", sizeof(struct Narty));  
    head.cena = 1500;  
    head.rok = 2017;  
    strcpy(head.model, "skitour");  
    blizard.rok = 2012; blizard.cena = 1359;  
    strcpy (blizard.model, "gigant");  
    mojenar->rok = 2019;  
    printf("Moje narty:\n cena =%.2lf, rok: %d, model: %s\n",\  
mojenar->cena, mojenar->rok, mojenar->model);  
    printf("Narty blizard:\n cena =%.2lf, rok: %d, model: %s\n",\  
blizard.cena, blizard.rok, blizard.model);  
    printf("Narty head:\n cena =%.2lf, rok: %d, model: %s\n",\  
head.cena, head.rok, head.model);  
    return 0; }
```

Definiowanie struktury globalnej.

*Powołanie obiektów z jednoczesną inicjalizacją egzemplarza **fischer**.*

Obiekt reprezentowany przez nazwę wskaźnika.

```
Rozmiar obiektow struktuty 'Narty'=24 bajty  
Moje narty:  
cena =1299.00, rok: 2019, model: slalom  
Narty blizard:  
cena =1359.00, rok: 2012, model: gigant  
Narty head:  
cena =1500.00, rok: 2017, model: skitour
```

*Rozmiar 24 bajty,
wielokrotność 8.
Z sumy 22 bajty,
2 bajty nieużywane,
tzw. **padding**.*

11.2. Struktury, **zagnieżdżenie**

W definiowanej strukturze polem może być inna struktura. Definiowanie takich struktur może zostać zrealizowane na dwa sposoby:

```
struct pierwsza {  
    char name[20];  
    char city[40];  
    long id; };  
  
struct druga{  
    struct pierwsza obiekt1;  
    char country[10];  
    int year_month[10][12];  
} obywatel1;
```

```
struct czwarta{  
    struct pierwsza *obiekt2;  
    char country[10];  
    int year_month[10][12];  
    char city;  
} *obywatel2;
```

Dostęp do pól struktury zagnieżdżonej realizowany jest przez podwójny operator **.**, lub odpowiednio operatory strzałek **->** lub ich kombinację w zależności w jaki sposób zostały powołane obiekty do poszczególnych struktur, np.: **obywatel1.obiekt1.id = 2034;**

obywatel2->obiekt2->id = 2034;

11.2. Struktury, **zagnieżdżenie**, p11-2d

Koszt tygodniowego wynajęcia sprzętu i instruktora w roku 2020 w marcu.

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

```
int main (void)
```

```
{struct Narty { Definiowania 1. etapu struktury.
```

```
    double cena;
```

```
    int rok;
```

```
    char model[10]; };
```

```
    struct Wyjazd{
```

```
        struct Narty head;
```

```
        double buty;
```

```
        int year_month[2][4]; } inst1;
```

```
inst1.year_month[2][3] = 200;
```

```
inst1.buty = 100;
```

```
inst1.head.cena = 120;
```

```
inst1.head.rok = 2017;
```

```
strcpy(inst1.head.model, "slalom");
```

```
printf ("Obiekt inst1:\n narty Head, typ: %s,\n rok: %d, cena = %.2lf\n"
```

```
, inst1.head.model, inst1.head.rok, inst1.head.cena);
```

```
printf (" buty = %.2lf, instruktor = %d\n", inst1.buty, inst1.year_month[2][3]);
```

```
return 0;}
```

*Definiowanie nowej
struktury
z zagnieżdżeniem
we wcześniejszej.*

Obiekt inst1:

narty Head, typ: slalom,

rok: 2017, cena = 120.00

buty = 100.00, instruktor = 200

11.2. Struktury, **typ strukturalny funkcji**, p11-2e

Definiowane funkcje własne mogą mieć typ strukturalny. W ten sposób przez instrukcję return można zwrócić wartość obiektu strukturalnego. Typ strukturalny może być również argumentem funkcji.

```
struct daneos {  
    char imie[15];  
    char nazwisko[25];  
struct daneos wczytaj1();  
void wyswietl1(struct daneos os);  
void wczytaj2(struct daneos *wsk);  
void wyswietl2(struct daneos *wsk);  
int main()  
{ struct daneos osoba, osoba2;  
printf("Osoba 1\n");  
    osoba=wczytaj1();  
    wyswietl1(osoba);  
printf("Osoba 2\n");  
    wczytaj2(&osoba2);  
    wyswietl2(&osoba2);  
return 0; }
```

```
struct daneos wczytaj1()  
{ struct daneos os;  
printf("Podaj imie: "); gets(os.imie);  
printf("Podaj nazwisko: "); gets(os.nazwisko);  
return os; }  
  
void wyswietl1(struct daneos os)  
{ printf("+ %s %s \n", os.imie, os.nazwisko);}
```

```
Osoba 1  
Podaj imie: Agnieszka  
Podaj nazwisko: Kloc  
+ Agnieszka Kloc  
Osoba 2  
Podaj imie: Stanislaw  
Podaj nazwisko: Pelka  
+ Stanislaw Pelka
```

11.2. Struktury, **typ strukturalny funkcji**, p11-2e

Wczytywanie danych **Osoby 2** zrealizowano wykorzystując argument typu strukturalnego z elementem wskazanym przez wskaźnik. W tym przypadku wartości są przekazane przez wskaźnik.

```
Osoba 1
Podaj imie: Agnieszka
Podaj nazwisko: Kloc
+ Agnieszka Kloc
Osoba 2
Podaj imie: Stanislaw
Podaj nazwisko: Pelka
+ Stanislaw Pelka
```

```
void wczytaj2(struct daneos *wsk)
{ printf("Podaj imie: "); gets(wsk->imie);
  printf("Podaj nazwisko: ");
  gets(wsk->nazwisko); }

void wyswietl2(struct daneos *wsk)
{ printf("+ %s %s \n", wsk->imie, wsk->nazwisko); }
```

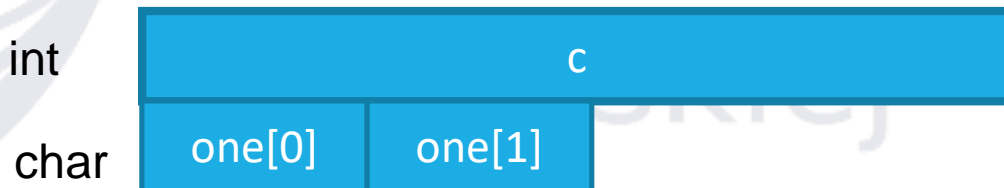
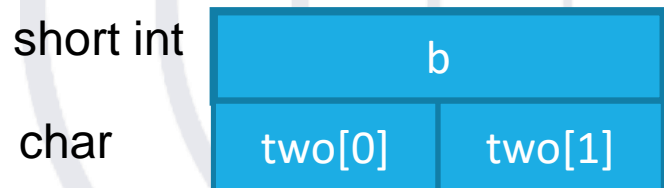
11.3. Unie, **union**

Unie są typem definiowanym przez użytkownika będący zestawem heterogenicznych zmiennych.

Zmienne **nakładają się na siebie zajmując** to samo miejsce w pamięci (rozpoczynają zajmowanie miejsca w tym samym miejscu).

Elementy składowe **unii** są rozmieszczone w pamięci **równolegle**.

Rozmiar unii jest rozmiarem jej największego typu składowego.



**Wykorzystanie
pamięci przez unie.**

W tym przypadku unia zajmuje więcej miejsca. Unia ma rozmiar jak największa zmienna **c (4 bajty)**

11.3. Unie, **union**

W unii wszystkie składowe obiektu umieszczane są pod tym samym adresem. Zatem w każdej chwili dostępna jest tylko jedna składowa. Powołanie do życia **unii** jest dwuetapowym procesem:

- zdefiniowanie typu unijnego,
- stworzenie konkretnego egzemplarza.

Etap 1:

```
union {  
    short beta;  
    unsigned char two[2];  
};
```

Etap 2:

```
union myunion;
```

Połączenie etapów

```
union {  
    short beta;  
    unsigned char two[2];  
} myunion;
```

Dostęp bezpośredni zapewnia **operator kropki** .

W przypadku użycia wskaźników **operator strzałki** ->

Przypisanie do składowej **beta** obiektu **myunion** zamaże poprzednią wartość tej składowej oraz poprzednią wartość składowej **two[2]**.

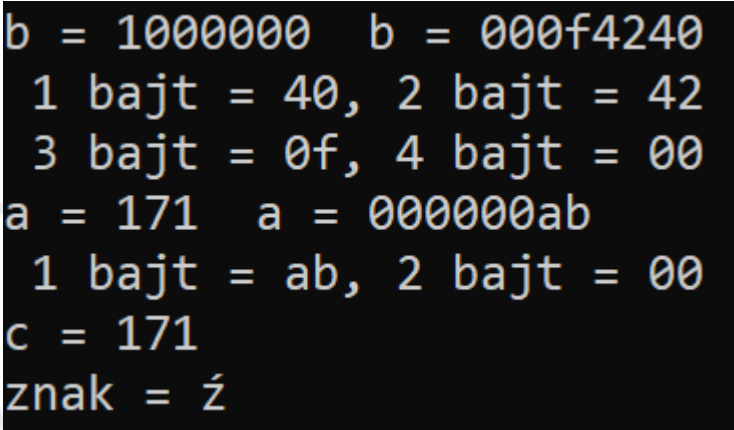
11.3. Unie, **union**, p11-3

Zdefiniowanie dwóch unii.

```
int main (void){  
    union uint{  
        int b;  
        unsigned char two[4];  
    } myunion;  
    union ushort{  
        short a; unsigned char one[2];  
        int c; char znak;  
    } sunion;  
    myunion.b = 1000000; Wstawiono wartość  
    printf ("b = %d ", myunion.b);  
    printf ("b = %8.8x\n 1 bajt = %2.2x, \  
2 bajt = %2.2x\n 3 bajt = %2.2x, 4 bajt = %2.2x\n", \  
myunion.b, myunion.two[0], myunion.two[1], \  
myunion.two[2], myunion.two[3]);  
    sunion.a = 171; Wstawiono wartość  
    printf ("a = %d ", sunion.a); printf ("a = %8.8x\n 1 bajt = %2.2x, 2 bajt = \  
%2.2x\n", sunion.a, sunion.one[0], sunion.one[1]);  
    printf ("c = %d\n",sunion.c); printf ("znak = %c\n",sunion.znak);  
    return 0; }
```

*Zdefiniowano dwie unie
i utworzono egzemplarze.*

*Unia spowodowała
wypełnienie tablic two
oraz one
(te same miejsca w pamięci)*



The screenshot shows the output of the C program. It displays the memory addresses and values for the union variables. For 'myunion', it shows 'b = 1000000' (000f4240) and the four bytes of the 'two' array: 40, 42, 0f, 00. For 'sunion', it shows 'a = 171' (000000ab) and the two bytes of the 'one' array: ab, 00. It also shows 'c = 171' and 'znak = ż'.

11.4. Pola bitowe, **struct**

Pola bitowe to zdefiniowanie zestawu danych, w których każdy z elementów (pole bitowe) **ma rozmiar określony w pojedynczych bitach**. Przykładowo umożliwia to podział zmiennej typu całkowitego (ze znakiem lub bez znaku) na segmenty o rozmiarach określone w bitach. Poszczególnym segmentom nadajemy nazwę (mogą istnieć pola bitowe bez nazwy).

Definiowanie pola bitowego:

```
struct nazwa_typu bitowego
{
    typ nazwa1: dlugosc1;
    typ nazwa2: dlugosc1;
    ...
    typ nazwaN: dlugosc1;
};
```

Możemy zastosować wersję dwuetapową lub jednoetapową. Należy pamiętać, aby powołać konkretny egzemplarz pola bitowego.

11.4. Pola bitowe, **struct**

```
struct byte {  
    char a: 1;  
    char b: 1;  
} bit;
```

Dostęp do pól bitowych:

- przy zastosowaniu zmiennych – za pomocą operatora kropki **.**
- przy adresowaniu wskaźników – operator strzałki **->**

Pamiętaj!

Nie można na siebie nakładać różnych pól (zazębiać).

Nienazwane pola bitowe służą do wyrównania obszaru pamięci i nie ma do nich dostępu.

11.4. Pola bitowe, **struct**, p11-4

Konwersja kodu ASCII znaku w postać dwójkową. M.M.Stabrowski

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

```
struct byte Pole bitowe do dekodowania binarnego.
```

```
{  
    char a: 1; // bit
```

```
    char b: 1;
```

```
    char c: 1;
```

```
    char d: 1;
```

```
    char e: 1;
```

```
    char f: 1;
```

```
    char g: 1;
```

```
    char h: 1; // bit
```

```
};
```

```
union bits
```

```
{
```

```
    char ch;
```

```
    struct byte bit;
```

```
}; ascii;
```

*Pole bitowe
zdefiniowane
jako zmienna
globalna.*

*Unia zdefiniowana
jako zmienna
globalna.*

Prototyp funkcji

```
void decode(union bits b);
```

```
int main (void)
```

```
{ do
```

```
    { if (ascii.ch != '\n')
```

```
        printf ("Enter character: ");
```

```
        ascii.ch = getchar();
```

Wprowadzenie znaku

```
        if (ascii.ch != '\n')
```

```
            { printf ("%c: ", ascii.ch);
```

```
                decode (ascii); }
```

Wywołanie funkcji

```
        } while (ascii.ch != 'q');
```

dekodującej

```
    return(0);
```

```
}
```

*Wyjście z programu
przez 'q'*

11.4. Pola bitowe, **struct**, p11-4

Konwersja kodu ASCII znaku w postać dwójkową. M.M.Stabrowski

void decode (union bits b) *Bity kodu ASCII*

```
{ if (b.bit.h) printf ("1 ");  
    else printf ("0 ");  
    if (b.bit.g) printf ("1 ");  
        else printf ("0 ");  
    if (b.bit.f) printf ("1 ");  
        else printf ("0 ");  
    if (b.bit.e) printf ("1 ");  
        else printf ("0 ");  
    if (b.bit.d) printf ("1 ");  
        else printf ("0 ");  
    if (b.bit.c) printf ("1 ");  
        else printf ("0 ");  
    if (b.bit.b) printf ("1 ");  
        else printf ("0 ");  
    if (b.bit.a) printf ("1 ");  
        else printf ("0 ");  
    printf ("  %#2x ", b.ch);  
    printf (" %3d\n", b.ch);  
}
```

h-a to poszczególne pola bitowe struktury bit

*b to egzemplarz unii bits. W naszym przypadku ten egzemplarz nosił nazwę **ascii**. Wywołanie **decode (ascii);***

Enter character: a		
a: 0 1 1 0 0 0 0 1	0x61	97
Enter character: A		
A: 0 1 0 0 0 0 0 1	0x41	65
Enter character: t		
t: 0 1 1 1 0 1 0 0	0x74	116
Enter character: T		
T: 0 1 0 1 0 1 0 0	0x54	84
Enter character: +		
+: 0 0 1 0 1 0 1 1	0x2b	43
Enter character: q		
q: 0 1 1 1 0 0 0 1	0x71	113

11.5. Wyliczenie enum

Typ wyliczeniowy enum służy do tworzenia zmiennych, które przyjmują wartości stałych całkowitych. W ten sposób uzyskujemy zbiór czytelnych nazw o wartościach całkowitych. Gdy nie zmienimy wartości to pierwsza ze zmiennych przyjmuje wartość 0, a kolejne są o 1 większe.

Definicja typu wyliczeniowego.

```
enum nazwa {  
    war_1, war_2, war_3, ... , war_n  
};  
enum nazwa obiekt1;
```

Konkretny egzemplarz.

11.5. Wyliczenie enum

W trakcie definiowania trybu wyliczeniowego można wprowadzić własne wartości do poszczególnych zmiennych. Należy pamiętać, że kolejna zmienna będzie miała wartość o 1 większą.

```
enum coin {  
penny, nickel=5, dime=10, quarter=25, half=50  
};
```

Przykład z programu:

*Definicja typu wyliczeniowego
o nazwie coin.*

```
enum coin {  
penny, nickel, dime, quarter, half  
};
```

```
enum coin money;
```

Konkretny egzemplarz

11.5. Wyliczenie enum, p11-5a

Tryb wyliczeniowy wykorzystany w instrukcji switch-case.

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

```
enum coin {
```

```
penny, nickel=5, dime=10,  
quarter=25, half_dollar=50,  
dollar=100 };
```

```
int main(void)
```

```
{enum coin money;
```

```
int k;
```

```
printf ("penny = %2d; nickel = \  
%2d\n", penny,nickel);
```

```
printf ("dime = %2d, quarter = \  
%2d;\n", dime,quarter);
```

```
printf ("half_dollar = %d, dollar = \  
100\n",half_dollar,dollar);
```

```
printf("Podaj wartosc monety\t");
```

```
scanf("%d",&k);
```

```
switch (k) {
```

```
case penny: printf ("penny\n"); break;
```

```
case nickel: printf ("nickel\n"); break;
```

```
case dime: printf ("dime\n"); break;
```

```
case quarter: printf ("quarter\n");
```

```
break;
```

```
case half_dollar: printf ("half_dollar\n");
```

```
break;
```

```
case dollar: printf ("dollar\n"); break;
```

```
default: printf ("unrecognized\n"); }
```

```
return 0; }
```

```
penny = 0; nickel = 5  
dime = 10, quarter = 25;  
half_dollar = 50, dollar = 100  
Podaj wartosc monety 25  
quarter
```

11.5. Wyliczenie enum, p11-5b

Tryb wyliczeniowy jako argument funkcji.

```
#include <stdio.h>
#include <stdio.h>
enum kolory {czerwony, pomarancz,
zolty, zielony, niebieski, blekitny,
fioletowy}; Konkretny egzemplarz
void paleta_barw (enum kolory kol);
int main(int argc, char *argv[])
{int k;
printf("Kolory palety barw\n");
printf("Wybierz pozycje koloru od 0 \
do 6\n");
scanf("%d", &k);
paleta_barw(k);
return 0;
}
```

```
Kolory palety barw
Wybierz pozycje koloru od 0 do 6
2
zolty - to cieply kolor
```

```
void paleta_barw (enum kolory kol)
{ switch (kol) {
case czerwony: printf("czerwony");break;
case pomarancz: printf("pomaranczowy");break;
case zolty: printf("zolty"); break;
case zielony: printf("zielony ");break;
case niebieski: printf("niebieski");break;
case blekitny: printf("blekitny");break;
case fioletowy: printf("fioletowy"); break;
default: printf("brak takiego koloru\n"); }
switch (kol) {
case czerwony:
case pomarancz:
case zolty: printf(" - to cieply kolor\n"); break;
case zielony:
case niebieski:
case blekitny:
case fioletowy: printf(" - to chlodny kolor\n");
break; } }
```

11.5. Wyliczenie enum, problemy

Brak jednoznaczności przy użyciu enum w instrukcji switch-case.
Definicja typu wyliczeniowego o nazwie dane.

```
enum dane{  
    jeden, dwa, trzy, cztery=-1, dalej1, dalej2  
} liczby;
```

W zaistniałej sytuacji zmienne przyjmą następujące wartości:

```
jeden = 0  
dwa = 1  
trzy = 2  
cztery = -1  
dalej1 = 0,  
dalej2 = 1
```

Tak więc dwie różne zmienne będą miały tą samą wartość.

11.6. Definicje nazwy typu **typedef**

Słowo kluczowe **typedef** służy do definiowania nowych nazw (identyfikatorom) istniejącym już typom. Należy pamiętać, że w ten sposób określamy tylko *nazwę* istniejącego typu, a *nie* tworzymy nowy typ.

W ten sposób można więc

- nadać samym typom danych nazwy związanych z ich wartością,
- tworzyć nowe nazwy typów danych uzależnionych od środowiska programistyczno-sprzętowego.
- zwięźle zapisywać definiowane egzemplarze **struktur** i **unii**.

Sposób użycia deklaracji **typedef**:

typedef nazwa_istniejacego_typu nowa_nazwa_typu;

11.6. Definicje nazwy typu **typedef**

Przykład użycia:

```
long NOWY_INT;
```

oznacza to, że zmienna `NOWY_INT` będzie typu `long`.

Po zastosowaniu deklaracji **typedef**:

```
typedef long NOWY_INT;
```

Uzyskujemy, że `NOWY_INT` jest nową nazwą typu `long`.

Tak więc możemy zastosować zapis:

```
NOWY_INT kasa;
```

co jest równoważne deklaracji:

```
long kasa;
```

W zakresie widoczności deklaracji **typedef**.

11.6. Definicje nazwy typu **typedef**, p11-6a

Specyfikator typedef użyty do wygodnego określania typu parametrów.

typedef int IN3[][2][2]; *Definicja typedef.*

int fun (IN3 t); *Definicja prototypu*

int main()

{ IN3 in3 = { {{4,3},{2,1}}, {{7,8},{5,6}} };

int **max** = fun(in3);

printf("Wartosc max = %d\n", **max**);

int roz=sizeof(in3);

printf("Rozmiar tablicy = %d\n", roz);}

int fun(IN3 t) {

int max = t[0][0][0];

int k, j, i;

for (k = 0; k < 2; ++k)

for (j = 0; j < 2; ++j)

for (i = 0; i < 2; ++i)

if (t[k][j][i] > max) max = t[k][j][i];

return **max**; }

Wartosc max = 8
Rozmiar tablicy = 32

11.6. Definicje nazwy typu **typedef**

Przykład 1. Zmiana nazwy trybu wyliczeniowego **enum** na prostszą nazwę.

```
typedef paleta_barw tecza;
```

Przykład 2. Zmiana nazwy typu **unsigned int** na nową ogólniejszą nazwę typu **size_t**.

Typ **size_t** jest zdefiniowany w nagłówku **stddef.h**, jako alias do liczby całkowitej bez znaku

```
typedef unsigned int size_t;
```

Użycie **size_t** może poprawić przenośność, wydajność i czytelność kodu.

Materiały zostały opracowane w ramach projektu
„Zintegrowany Program Rozwoju Politechniki Lubelskiej – część druga”,
umowa nr **POWR.03.05.00-00-Z060/18-00**
w ramach Programu Operacyjnego Wiedza Edukacja Rozwój 2014-2020
współfinansowanego ze środków Europejskiego Funduszu Społecznego

NAZWA PRZEDMIOTU
Programowanie strukturalne

Temat wykładu 12.

**Podstawowe operacje plikowe, znaczniki trybu dostępu do plików.
Binarne operacje plikowe. Dostęp swobodny do plików.**

dr hab. inż. Jerzy Montusiewicz, prof. PL

12. Agenda

12.1. Podstawowe operacje plikowe.

12.2. Argumenty funkcji main().

12.3. Znaczniki trybu dostępu do plików.

12.4. Przykłady użycia operacji plikowych.

12.5. Binarne operacje plikowe.

12.6. Dostęp swobodny do plików.

12.1. Podstawowe operacje plikowe

Programy nie działają bezpośrednio na zawartości plików, które mogą reprezentować różny tryb dostępu (np.: sekwencyjny, swobodny, do odczytu, itd.).

Wprowadzone strumienie widziane są bezpośrednio z programu:

- **strumień** – to logiczny abstrakcyjny kanał transmisji informacji **do/z** pliku.
- **pliki** – to aktualne urządzenie lub obiekt przechowujący informację.

Strumienie dzielimy na:

- **tekstowe** – w czasie transmisji informacji dokonują pewnej konwersji, np. znak nowego wiersza),
- **binarne** – bez jakichkolwiek konwersji).

12.1. Podstawowe operacje plikowe

Dotychczas korzystaliśmy ze **stdin** oraz **stdout**.

Te strumienie są **buforowane**, przy awarii zawartość buforów może być stracona.

Niebuforowany strumień **stderr** do wypisywania błędów.

Najlepiej **stosować jawne otwarcie** strumienia i powiązanie go z konkretnym plikiem.

Dobra praktyka nakazuje **zamknięcie niepotrzebnych strumieni**.

Zamknięcie powoduje:

- zwolnienie zasobów użytkowanych,
- zapobiega utracie buforowanej informacji.

12.1. Podstawowe operacje plikowe

Otwarcie strumienia i powiązanie go z plikiem:

FILE *`fopen` (const char *`path`, const char *`mode`);

- funkcja **`fopen()`** zwraca wskaźnik do predefiniowanej struktury **FILE** (nie powinniśmy jej modyfikować),
- wskaźnik ten jest używany przez funkcje przetwarzające zawartość pliku,
- argument ***`path`** jest wskaźnikiem do łańcucha znakowego zawierającego nazwę pliku (nazwa ścieżki dostępu do pliku),
- argument ***`mode`** jest łańcuchem znakowym określający tryb dostępu do pliku (tabela w podrozdziale 12.3, slajd 14).

12.1. Podstawowe operacje plikowe, rodzina **get()**

Do realizacji operacji **znakowego nieformatowanego odczytu** informacji z pliku stosujemy następujące funkcje:

int fgetc (FILE *stream);

int getc (FILE *stream);

int ungetc (int c, FILE *stream);

char *fgets (char *s, int size, FILE *stream);

Funkcje **fgetc()** i **getc()** działają jak funkcja **getchar()**, ale ogólniej.

W funkcji **getchar()** nie podajemy nazwy strumienia (domyślnie **stdin**). Wymienione funkcje odczytują dane ze wskazanego strumienia (pliku).

Pamiętaj! Funkcja **getchar()** jest równoważna funkcji **getc (stdin)**.

12.1. Podstawowe operacje plikowe, rodzina **get()**

Funkcja **getc()** jest najczęściej zrealizowana jako makro.

int ungetc (int c, FILE *stream);

Funkcja **ungetc()** odkłada znak c (rzutowanie na unsigned char) z powrotem do strumienia **stream** i dlatego znak c może być dostępny dla kolejnej operacji odczytu (zagwarantowane jest jedno odłożenie).

char *fgets (char *s, int size, FILE *stream);

Funkcja **fgets()** ma dodatkowy argument **size** (maksymalna liczba wczytywanych znaków), zapobiega przepełnieniu bufora wskazanego przez *s, np.:

```
char buf[64];  
fgets(buf, sizeof buf, stdin);
```

12.1. Podstawowe operacje plikowe, rodzina put()

Plikowe funkcje wyjścia:

int fputc (int c, FILE *stream);

int putc (int c, FILE *stream);

int fputs (const char *s, FILE *stream);

Funkcje **fputc()** i **putc()** działają jak funkcja **putchar()**.

Funkcja **putc()** może być realizowana jako makro.

Funkcja **fputs()** jest analogiczna jako **puts()**, ale nie zapisuje do pliku końcowego znaku `"\0"`.

Funkcja **puts()** wysyła łańcuch jedynie na standardowe wyjście (**stdout**) oraz znak nowej linii (`\n`).

12.1. Podstawowe operacje plikowe

Funkcje obsługujące dotarcie do końca pliku:

int feof (FILE *stream);

int ferror (FILE *stream);

void clearerr (FILE *stream);

Funkcja **feof()** sprawdza znacznik końca pliku dla strumienia wskazanego przez **stream**, zwraca wartość niezerową jeśli jest on ustawiony.

Funkcja **ferror()** sprawdza znacznik błędu dla strumienia wskazanego przez **stream**, zwraca wartość niezerową jeśli jest on ustawiony.

Funkcja **clearerr()** usuwa znacznik końca pliku i błędu dla strumienia wskazanego przez **stream**.

12.2. Argumenty funkcji main()

W chwili uruchamiania programu (wersja .exe) można przekazać informacje poprzez argumenty funkcji main().

Wpisujemy nazwę pliku ze skompilowanym programem (wersja .exe) i kilka łańcuchów alfanumerycznych.

Musimy jednak wpisać argumenty funkcji main(), np.:

```
int main (int argc, char *argv[])
```

int argc

– liczba łańcuchów,

char *argv[]

– właściwie to tablica wskaźników do typów char (akceptowanie łańcuchów o dowolnej długości).

‘**c**’ od count, zaś ‘**v**’ od value.

```
int main(int argc, char **argv[])
```

12.2. Argumenty funkcji main(), p12-2a

Program main() z argumentami. Przy złej liczbie argumentów program nie zostanie uruchomiony w wersji .exe.

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

```
#include <stdlib.h>
```

```
int main (int argc, char *argv[])
```

```
{ if (argc != 2)      Bezwarunkowe wyjście z programu.
```

```
{ printf ("Usage: %s user_name\n", argv[0]);
```

```
    exit (99); }
```

```
printf ("Hello %s!\n", argv[1]);
```

```
system("pause");
```

```
return 0;
```

```
}
```

*Powtórne uruchomienie
z podaniem argumentów:*

p12-2a_arg_main.exe Jurek

*Pierwszy argument – nazwa
funkcji, drugi – stała łańcuchowa.*

Uruchomienie w środowisku Dev-cpp.

```
Usage: F:\dydaktyka\2019_projekt  
_nowa_informatyka\0_Programowani  
e_strukturalne_wyk\ady-1\program  
y_do_wyk\ad_w\W_12\p12-2a_arg_ma  
in.exe user_name
```

```
Hello Jurek!  
Press any key to continue
```

12.2. Argumenty funkcji main(), p12-2b

Program main() z argumentami. Przetwarzanie argumentu wiersza polecenia w postać liczbową, funkcja biblioteczna **atoi()** [ASCII to int].

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

```
#include <stdlib.h>
```

```
int main (int argc, char *argv[])
```

```
{int liczba;
```

```
if (argc != 2)
```

```
{ printf ("Usage: %s user_name\n", argv[0]);
```

```
exit (99); }
```

```
printf ("The argument as a char = %s\n", argv[1]);
```

```
liczba = atoi(argv[1]); Wywołanie funkcji atoi()
```

```
printf ("The argument as a value = %d\n", liczba);
```

```
getchar(); Wstrzymanie wykonania programu.
```

```
return 0;
```

```
} Ponowne uruchomienie programu:
```

p12-2b_atoi.exe 10

```
Usage: F:\dydaktyka\2019_projekt_no  
wa_informatyka\0_Programowanie stru  
kturalne_wyk\ady-1\programy do wyk  
ad\w\W_12\p12-2b_atoi.exe user_name
```

```
The argument as a char = 10  
The argument as a value = 10
```

12.2. Argumenty funkcji main(), 12-2c1

Program main() z argumentami. Dostęp wskaźnikowy.

```
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char **argv[])
{ if (argc != 2)
  {printf ("Usage: %s user_name\n", argv[0]);
   exit (99); }
  printf ("Hello %s\n", *(argv+1));
  system("pause");
  return 0;
}
```

Ponowne uruchomienie:

p12-2c1_wskaz.exe Informatyka

```
Hello Informatyka
Press any key to continue
```

**++argv lub *(argv++).*

*Gdy *(argv++) to wypisze nazwę aplikacji.*

Ponowne uruchomienie:

p12-2c1_wskaz-pp.exe Aga

```
Hello F:\dydaktyka\2019_projekt_nowa_informatyka\0_Programowanie_strukturalne_wyk\ady-1\programy_do_wyk\ad~w\W_12\p12-2c3_wskaz-pp.exe
```


12.3. Znaczniki trybu dostępu do plików

Znaczniki trybu dostępu do plików używane w funkcji **fopen()**.

Znacznik	Działanie
r	Otwarcie pliku tekstowego do odczytu. Strumień wskazuje początek pliku.
r+	Otwarcie pliku tekstowego do odczytu i zapisu. Strumień wskazuje początek pliku.
w	Usunięcie zawartości pliku lub utworzenie nowego pliku tekstowego do zapisu. Strumień wskazuje początek pliku.
w+	Otwarcie pliku tekstowego do odczytu i zapisu. Jeśli plik nie istnieje to zostanie utworzony. Gdy istnieje to jego zawartość zostanie usunięta. Strumień wskazuje początek pliku.
a	Otwarcie pliku tekstowego do dopisywania (zapisu na końcu pliku). Jeśli plik nie istnieje to zostanie utworzony. Strumień wskazuje na koniec pliku.
a+	Otwarcie pliku tekstowego do odczytu i dopisywania (zapisu na końcu pliku). Jeśli plik nie istnieje to zostanie utworzony. Strumień wskazuje na koniec pliku.

Dodanie znaku **"b"** (na końcu lub w środku) oznacza, że plik zostaje otwarty w trybie **dostępu binarnego**, np.: **"wb"**, **"wb+"**.

12.4. Przykłady użycia operacji plikowych, p12-4a

Zapis wartości do pliku. Dane i nazwę pliku wpisujemy z klawiatury.

```
int main (int argc, char *argv[])
{ FILE *fp;      Deklaracja wskaźnika do struktury FILE
  char ch;
  if (argc != 2) {printf ("Nie podano nazwy pliku wyjściowego!\n"); goto stop1;}
  if ((fp=fopen(argv[1], "w")) == NULL) { printf („Nie można otworzyć \
pliku.\n"); goto stop2; } Otwarcie dostępu do pliku.
  printf("Wpisz napis, zakończenie przez znak '$'\n");
  do { ch = getchar (); Wczytywanie znaków
      if (EOF == putc (ch, fp))
        { printf ("Błąd w pliku!\n"); break; } Zapisywanie znaku w pliku.
      } while (ch != '$');
  fclose (fp); Zamknięcie strumienia zapisu
  system("pause");
  stop1:
  stop2:
  return 0;
}
```

Tryb dostępu: "w"

Sprawdzenie otwarcia ==NULL

Wskaźnik fp użyty do otwarcia strumienia komunikacji z plikiem.

Uruchomienie w Dev-Cpp.

Nie podano nazwy pliku wyjściowego!

Uruchomienie z argumentem:
p12-4a.exe Inf_1

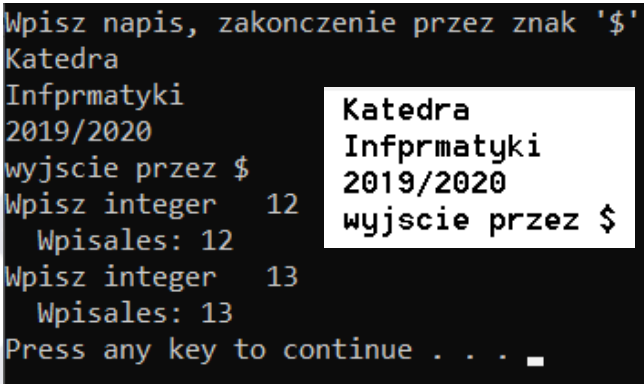
12.4. Przykłady użycia operacji plikowych, p12-4a

Zapis wartości do pliku. Dane i nazwę pliku wpisujemy z klawiatury.

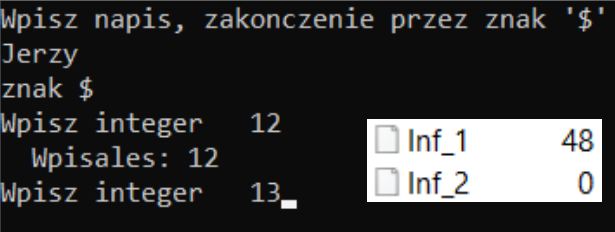
Sprawdzenie działania funkcji **fclose()**.

```
int main (int argc, char *argv[])
{ FILE *fp; char ch;
  if (argc != 2) {printf ("Nie podano nazwy pliku wyjsciowego!\n"); goto stop1;}
  if ((fp =fopen(argv[1], "w")) == NULL) { printf („Nie mozna otworzyć pliku.\n");
    goto stop2; } Otwarcie strumienia komunikacji z plikiem.
  printf("Wpisz napis, zakonczenie przez znak '$\n");
  do { ch = getchar ();
      if (EOF == putc (ch, fp))
      { printf ("Blad w pliku!\n"); break; }
      while (ch != '$');
// fclose (fp); Brak zamknięcie strumienia zapisu
  int a2,a3; printf("Wpisz integer\t"); scanf("%d", &a2);
  printf(" Wpisales: %d\n", a2);
  printf("Wpisz integer\t"); scanf("%d", &a3);
  printf(" Wpisales: %d\n". a3):
    system("pause");
  stop1:
  stop2:
  return 0; }
```

*Uruchomienie z aktywnym
fclose(): p12-4a.exe Inf_1*



*Uruchomienie bez fclose():
p12-4a.exe Inf_2
Przerwanie programu!*



12.4. Przykłady użycia operacji plikowych, p12-4b

Czytanie i zapisywanie danych w plikach. Program czyta dane z jednego pliku znacznik "r", przetwarza je i zapisuje do drugiego pliku znacznik "w".

- 1. Czytanie** pliku wejściowego wiersz po wierszu, ten plik musi istnieć.
- 2. Przetwarzanie** odczytanego wiersza. Program na początku dodaje numer kolejnego wiersza ze znakiem '·'.
- 3. Zapis** nowego wiersza. Zmodyfikowany wiersz zapisujemy do pliku wyjściowym, jeśli plik nie istnieje to zostanie utworzony.

Pamiętaj! Gdy plik istnieje to jego zawartość będzie wymazana, ("w").

Modyfikacja programu znacznik "a"

12.4. Przykłady użycia operacji plikowych, p12-4b

Czytanie i zapisywanie danych w plikach.

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

```
int main (int argc, char *argv[])
```

```
{ FILE *in, *out;      Deklaracja wskaźników do struktury FILE
```

```
→ char *fname_in, *fname_out; Deklaracja wskaźników nazw plików.
```

```
char in_line[255];      Bufor wejściowy.
```

```
int no_line;
```

```
if (argc < 3) { printf ("Uzycie: %s plik_we plik_wy\n", argv[0]); goto stop1; }
```

```
→ fname_in = argv[1]; Podstawienie adresów.
```

```
fname_out = argv[2];      Otwarcie pliku wejściowego i wyjściowego.
```

```
→ if ((in = fopen(fname_in, "r")) == NULL || ((out = fopen(fname_out, "w")) == NULL))  
{ printf ("Otwarcie pliku wej lub wyj niemożliwe!\n"); goto stop2; }
```

```
no_line = 1;      fgets() odczytuje znak końca linii.
```

```
while (fgets(in_line, 255, in) != NULL)      Odczyt pliku wiersz po wierszu.
```

```
{ fprintf(out, "%2.2d: %s", no_line, in_line);
```

```
→ no_line++; }      Zapis do pliku po modyfikacji,
```

```
stop1:      dodanie numeracji i ':'.  
stop2:
```

```
return 0;
```

```
}      Uruchomienie: p12-4b.exe Inf_1 Out_1
```

12.4. Przykłady użycia operacji plikowych, p12-4b

Wersja ze znacznikiem "w".

Uruchomienie: p12-4b_w.exe Inf_1 Out_1

1. Zawartość pliku wejściowego Inf_1:
2. Zawartość pliku wyjściowego Out_1 po uruchomieniu programu, odczyt, modyfikacja, zapis:

Plik Edytuj Opcje Kodowanie

Katedra
Infprmatyki
2019/2020
wyjscie przez \$

Plik Edytuj Opcje Kodowanie

01: Katedra
02: Infprmatyki
03: 2019/2020
04: wyjscie przez \$

Wersja ze znacznikiem "a".

Uruchomienie: p12-4b_a.exe Inf_1 Out_1

1. Zawartość pliku wejściowego Inf_1, tak jak poprzednio.
2. Początkowa zawartość pliku wyjściowego Out_1, z poprzedniego uruchomienia, odczyt, modyfikacja, dopisanie nowych informacji:

Plik Edytuj Opcje Kodowanie

01: Katedra
02: Infprmatyki
03: 2019/2020
04: wyjscie przez \$
01: Katedra
02: Infprmatyki
03: 2019/2020
04: wyjscie przez \$

12.5. Binarne operacje plikowe

Binarne operacje plikowe wykonywane są na plikach otwartych z flagą dostępu "b" (konieczne w środowisku Windows) i korzystające z funkcji odczytu i zapisu bloków danych.

size_t fread(void *ptr, size_t size, size_t l_blk, FILE *strm);

Funkcja **fread()** odczytuje ze strumienia wskazanego przez *strm do bufora wskazanego przez *ptr, l_blk bloków danych każdy o rozmiarze size bajtów.

size_t fwrite(const void *ptr, size_t size, size_t l_blk, FILE *strm);

Funkcja **fwrite()** zapisuje do strumienia wskazywanego przez *strm, l_blk bloków danych każdy o rozmiarze size bajtów, pobierając je z bufora wskazanego przez *ptr.

Funkcje te zwracają liczbę bloków (zapis/odczyt) a nie liczbę bajtów.

size_t – alias typu całkowitego bez znaku (więcej wykład 14, slaid 17).

12.5. Binarne operacje plikowe, p12-5a

Program otwiera dostęp do istniejącego pliku w trybie binarnym "wb".

```
#include <stdio.h>
#define N 20
int main (int argc, char *argv[])
{ FILE *fp; float dane[N]; int i;
  if (argc != 2) { printf ("Nie podano nazwy pliku wejscowego\n"); goto stop1; }
  if ((fp = fopen(argv[1], "wb")) == NULL)
  { printf ("Nie mozna otworzyc pliku.\n"); goto stop2; }
  for (i=0; i<N; i++) dane[i] = (float)i+1;
  if (fwrite (dane, sizeof(dane), 1, fp) !=1)
  { printf("Blad w pliku!\n"); goto stop3; }
  fclose (fp);
stop1:
stop2:
stop3:
  return 0;
}
```

Wypełniono tablicę w pętli for

Funkcja fwrite() – zapis w trybie binarnym.

Uruchomienie spod kompilatora.

Nie podano nazwy pliku wejscowego.

Uruchomienie ponowne: p12-5a_fwrite.exe bin_out.

bin_out

Rozmiar utworzonego pliku: 20 elementów x 4 bajty.

80

12.5. Binarne operacje plikowe, p12-5b

Program otwiera dostęp do istniejącego pliku w trybie binarnym "rb".

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

```
#define N 20
```

```
int main (int argc, char *argv[])
```

```
{ FILE *fp; float tab[N]; int i;
```

```
if (argc != 2) { printf ("Nie podano nazwy pliku wejsciowego.\n"); goto stop1; }
```

```
if ((fp = fopen(argv[1], "rb")) == NULL) Pliki wejściowy.
```

```
{ printf ("Nie mozna otworzyc pliku.\n"); goto stop2; }
```

```
if (fread (tab, sizeof(tab), 1, fp) !=1) Odczytywanie pliku. Przeczytano jeden blok.  
{ if (feof(fp) !=EOF) { printf("Bład odczytu pliku!\n"); goto stop3; } }
```

```
fclose (fp);
```

```
for (i=0; i<N; i++) { printf ("%5.2f  ", tab[i]);
```

```
if ((i+1)%5 == 0) printf ("\n"); }
```

```
stop1:
```

```
stop2:
```

```
stop3:
```

```
system ("pause");
```

```
return 0; }
```

Funkcja fread() – odczyt w trybie binarnym "rb".

*Uruchomienie ponowne:
p12-5a_fread.exe bin_out*

1.00	2.00	3.00	4.00	5.00
6.00	7.00	8.00	9.00	10.00
11.00	12.00	13.00	14.00	15.00
16.00	17.00	18.00	19.00	20.00

Odczyt w trybie binarnym i wyprowadzenie na monitor w formacie f.

12.6. Dostęp swobodny do plików

Zapis i odczyt z pliku odbywa się w miejscu określonym przez wskaźnik pozycji w pliku.

Zawartość wskaźnika może być zmieniana w sposób jawny. Pozwala to więc na przemieszczanie się wewnątrz pliku i dokonywać operacji w dowolnym miejscu pliku (**tryb dostępu swobodnego**).

Dostępny jest jeden wskaźnik dla danego pliku, określający miejsce zapisu i odczytu.

Istnieją następujące funkcje biblioteczne:

```
int fseek (FILE *strm, long offset, int origin);
```

```
long ftell (FILE *strm);
```

```
void rewind (FILE *strm);
```

12.6. Dostęp swobodny do plików

Funkcja **fseek()** ustawia wskaźnik pozycji pliku dla strumienia ***strm**. Nowa pozycja określona w bajtach jest odległością **offset**, mierzona od bazy określonej argumentem **origin**.

Ostatnim argumentem powinno być jedno z makr:

SEEK_SET, SEEK_CUR lub **SEEK_END**

- **SEEK_SET** – oznacza początek pliku,
- **SEEK_CUR** – oznacza pozycję bieżącą pliku,
- **SEEK_END** – oznacza koniec pliku.

Funkcja **ftell()** odczytuje i zwraca bieżącą wartość wskaźnika pliku obsługiwanego przez strumień ***strm**.

Funkcja **rewind()** ustawia wskaźnik pozycji w pliku na początku pliku, co jest równoważne operacji:

(void)fseek (strm, OL, SEEK_SET);

12.6. Dostęp swobodny do plików, p12-6a

Program zmienia zawartość we wskazanym miejscu. Użycie argumentów `SEEK_SET` i `SEEK_END`. Tryb dostępu do pliku `"r+"`: czytanie i zapis.

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

```
int main (int argc, char *argv[])
```

```
{ FILE *fptr;
```

```
if (argc != 3) { printf ("Uzycie: %s plik przesuniecie\n", argv[0]); return 9; }
```

```
if ((fptr = fopen(argv[1], "r+")) == NULL)
```

```
{ printf ("Nie mozna otworzyc pliku.\n"); system ("pause"); return 1; }
```

```
fseek (fptr, atoi (argv[2]), SEEK_SET);
```

```
putc ('^', fptr);
```

```
fseek (fptr, -atoi(argv[2]), SEEK_END); putc ('\\', fptr);
```

```
fseek (fptr, -atoi(argv[2]), SEEK_END); putc ('\\', fptr);
```

```
fclose (fptr);
```

```
system ("pause");
```

```
return 0;
```

```
}
```

Uruchomienie z wersji .exe:

p12-6a.exe Inf_1 0

argv[2] – wprowadzono wartość 0.

Zapis pliku na pozycji liczonej od początku.

atoi(argv[2]) – zamienia zmienną char na wartość liczbową.

Zapis w pliku na pozycji liczonej od końca.

Przesuniecie jest liczbą ujemną.

```
^atedra
Infprmatyki
2019/2020
wyjscie przez $
\\
```

12.6. Dostęp swobodny do plików, p12-6b

Program zmienia zawartość we wskazanym miejscu. Próba wpisania kilku znaków. Użycie argumentów **SEEK_SET** i **SEEK_END**. Tryb dostępu do pliku **"r+"**: czytanie, zapis.

Fragment kodu programu.

```
/* Zapis pliku na pozycji liczonej od początku. */  
fseek (fptr, atoi (argv[2]), SEEK_SET);  
putc ('-^-', fptr); // wpisano 3 znaki
```

Tryb "r+" nie kasuje wnętrza pliku, program modyfikuje plik.

Uruchomienie z wersji .exe:
p12-6b.exe Inf_1 3

Plik Edytuj Opcje Kodowanie
Kat-dra
Infprmatyki
2019/2020
wyjscie przez \

Pamiętaj! Można wpisać tylko jeden znak ujęty w apostrofy.

Materiały zostały opracowane w ramach projektu
„Zintegrowany Program Rozwoju Politechniki Lubelskiej – część druga”,
umowa nr **POWR.03.05.00-00-Z060/18-00**
w ramach Programu Operacyjnego Wiedza Edukacja Rozwój 2014-2020
współfinansowanego ze środków Europejskiego Funduszu Społecznego



**Fundusze
Europejskie**
Wiedza Edukacja Rozwój



**Rzeczpospolita
Polska**

Unia Europejska
Europejski Fundusz Społeczny



NAZWA PRZEDMIOTU
Programowanie strukturalne

Temat wykładu 13.

**Standardowe funkcje wejścia-wyjścia nieformatowane i formatowane.
Specyfikacje kodów dla grupy printf(), scanf(), modyfikatory, flagi.**

dr hab. inż. Jerzy Montusiewicz, prof. PL

13. Agenda

- 13.1. Standardowe funkcje wejścia nieformatowane.**
- 13.2. Standardowe funkcje wejścia formatowane.**
- 13.3. Standardowe funkcje wyjścia nieformatowane.**
- 13.4. Standardowe funkcje wyjścia formatowane.**
- 13.5. Specyfikacje kodów dla grupy printf().**
- 13.6. Specyfikacje kodów dla grupy scanf().**
- 13.7. Inne operacje plikowe.**

13.1. Standardowe funkcje wejścia nieformatowane

W języku C operacje wejścia-wyjścia realizowane są za pomocą funkcji bibliotecznych.

Podział:

- **standardowe** (domyślnego wejścia-wyjścia; **stdin/stdout**),
- wejścia-wyjścia **plikowe** (to było w wykładzie 12).

Standardowe wejścia-wyjścia są w zasadzie szczególnym przypadkiem wejścia-wyjścia plikowego.

Funkcje nieformatowanego standardowego wejścia-wyjścia służą do wczytania lub wypisania pojedynczych znaków lub łańcuchów znakowych napisów.

Obsługiwane przez plik nagłówkowy **stdio.h**.

13.1. Standardowe funkcje wejścia, nieformatowane

Podstawowe funkcje wejścia (funkcje biblioteki C):

char *gets (char *str);

- odczytuje **pojedynczą linię** ze **stdin** (klawiatura) i zapisuje go do bufora wskazanego jako argument (*str),
- zatrzymuje się, gdy odczyta **znak nowej linii** (\n) lub gdy zostanie osiągnięty koniec pliku EOF (co nastąpi wcześniej?), np.:

char str[50]; gets(str);

- nie kontroluje przepełnienia bufora.

int getchar(void);

- dostaje znak (unsigned char) ze standardowego wejścia (**stdin**),
- funkcja oczekuje na znak nowego wiersza (klawisz **Enter**),
- można implementować jako makro, gdy dokonuje odczytu więcej niż jeden raz,
- funkcja ta jest równoważna **getc** ze **stdin** jako argument.

13.2. Standardowe funkcje wejścia formatowane

Standardowe wejście formatowane jest obsługiwane przez rodzinę funkcji **scan(f)** (funkcje biblioteki C), najczęściej to:

```
int scanf (const char *k_for, arg_list);
```

arg_list – lista zmiennych (rozdzielona przecinkami) przetwarzana przez kody formatu umieszczone w stałej napisowej **k_for**.

Pamiętaj! Sposób oddzielania kodów formatujących decyduje o sposobie oddzielania zmiennych przy wprowadzaniu wartości wpisywanych z klawiatury, np.: `%d,%d` implikuje `7,13` ; `%d/%d` wymusza `7/13`, zaś `%d%d` oznacza, że należy wpisać `7 13` (spacja).

```
int sscanf (const char *str, const char *k_for, arg_list);
```

Funkcja pozwala na odczytywanie informacji z łańcucha znakowego **str**.

13.2. Standardowe funkcje wejścia formatowane, p13-2a

Program czyta z łańcucha znaków. Przykład użycia funkcji `sscanf()` – formatowane czytanie.

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

```
int main ()
```

*Definicja zmiennej wskaźnikowej **string** i inicjalizacja.*

```
{ char *string = "int= 12 double= 77.12";
```

```
char inf1[12], inf2[12];
```

```
int ival; double dval;
```

```
puts ("\nPierwotna zawartosc lancucha:\n");
```

Pobranie wartości ze zmiennej.

```
puts (string); Wypisanie jej wartości.
```

```
sscanf (string, "%s %d %s %lf\n", inf1, &ival, inf2, &dval);
```

```
printf ("Informacje odczytane z lancucha:\n");
```

```
printf ("%s %d\n", inf1, ival);
```

```
printf ("%s %6.3lf\n", inf2, dval);
```

```
system("pause");
```

```
return 0;
```

```
} Flaga 'l' w 'lf' zmienia kod z float na double.
```

Pierwotna zawartosc lancucha:

```
int= 12 double= 77.12
```

```
Informacje odczytane z lancucha:
```

```
int= 12
```

```
double= 77.120
```

13.3. Standardowe funkcje wyjścia nieformatowane, p13-3a

Podstawowe funkcje wyjścia (funkcje biblioteki C):

int putchar (int c);

- funkcja wysyła na standardowe wyjście **stdout** (monitor) **znak c**, rzutowany na unsigned char, a następnie znak nowej linii,
- funkcja zwraca liczbę nieujemną w przypadku sukcesu,
- w przypadku błędu zwraca wartość EOF,
- można implementować jako makro, wykonywanie operacji wyjścia więcej niż jeden raz.

int puts (const char *s);

- funkcja wysyła na standardowe wyjście **napis s**, a następnie znak nowej linii,
- funkcja zwraca liczbę nieujemną w przypadku sukcesu,
- w przypadku błędu zwraca wartość EOF.

13.4. Standardowe funkcje wyjścia formatowane

Standardowe wyjście formatowane jest obsługiwane przez rodzinę funkcji **print()**. Funkcje biblioteki C.

int printf (const char *k_for, arg_list);

arg_list – opcjonalna lista zmiennych (rozdzielona przecinkami) przetwarzana przez kody formatu umieszczone w stałej napisowej **k_for**.

Funkcja do wyprowadzania informacji do dowolnego pliku (łańcuch znakowy **str**), a nie tylko na standardowe wyjście (stdout):

int sprintf (char *str, const char * k_for, arg_list);

Funkcja, która kontroluje maksymalną liczbę znaków (**size_c**) zapisywanych w łańcuchu znakowym (**str**):

int snprintf (char *str, size_c, const char * k_for, arg_list);

13.4. Standardowe funkcje wyjścia formatowane, p13-4a

Program zapisuje do tablicy znakowej. Przykład użycia funkcji **sprintf()** – formatowany zapis.

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

```
int main ()
```

```
{ char *string = "Janek ma narty firmy Fischer.\n";
```

```
char tab[80]; int ival = 2018; double dval = 14.8e+2;
```

```
printf("++ Zawartosc tablicy - funkcja sprintf()\n");
```

```
sprintf (tab, "%sRok produkcji nart: %d,\ncena zakupu nart: %0.2f \
```

```
PZL", string, ival, dval);
```

```
puts (tab); printf("\n");
```

```
printf("-- Wydruk klasyczny\  
- funkcja printf()\n");
```

```
printf ("%s Rok produkcji\  
nart: %d,\ncena zakupu nart:  
%4.2e PZL", string, ival, dval);
```

```
return 0;
```

```
}
```

*Przeniesienie
wartości do
tablicy **tab[]**.*

```
++ Zawartosc tablicy - funkcja sprintf()  
Janek ma narty firmy Fischer.  
Rok produkcji nart: 2018,  
cena zakupu nart: 1480.00 PZL
```

```
-- Wydruk klasyczny - funkcja printf()  
Janek ma narty firmy Fischer.  
Rok produkcji nart: 2018,  
cena zakupu nart: 1.48e+003 PZL
```

13.5. Specyfikacje kodów dla grupy **printf()**

Wybrane specyfikacje kodu formatu w funkcjach grupy **printf()**.

W kodzie formatu można podać:

- informację liczbową o szerokości pola (np.: %6d),
- dokładności liczby (liczba cyfr po kropce dziesiętnej (np.: %6.2f),
- gdy szerokość pola jest za duża to uzupełniana jest spacjami..

Kod	Format
d, i	Argument typu int jest konwertowany w postać dziesiętną ze znakiem. Domyślna dokładność to 1.
O,u,x,X	Argument typu unsigned int jest konwertowany w postać oktalną (o), dziesiętną (u), szesnastkową (x – małe, X – duże litery) – bez znaku.
e, E	Argument zmiennoprzecinkowy jest zaokrąglany i konwertowany w postać [-]d.dde±dd . Przed kropką jedna cyfra, po kropce tyle cyfr jaka jest dokładność (domyślnie 6). Wykładnik poprzedzony literą e lub E , ma zawsze dwie cyfry.

13.5. Specyfikacje kodów dla grupy **printf()**

Wybrane specyfikacje kodu formatu w funkcjach grupy **printf()**.

Kod	Format
f, F	Argument zmiennoprzecinkowy jest zaokrąglany i konwertowany w postać [-]ddd.ddd . Po kropce tyle cyfr ile wynosi dokładność (domyślnie 6). Gdy dokładność jest zerem – nie jest wypisywana kropka dziesiętna. Gdy występuje kropka jest przed nią przynajmniej jedna cyfra.
g, G	Argument zmiennoprzecinkowy jest konwertowany zgodnie ze specyfikacją e, E lub f, F . Specyfikacja e jest wybierana, gdy wykładnik jest mniejszy od -4 lub większy od podanej dokładności reprezentacji.
a, A	W standardzie C99 argument zmiennoprzecinkowy jest konwertowany w postać szesnastkową typu [-]0xh.hhhhp±d z separatorem wykładnika p lub P . Przed kropką wypisywana jest jedna cyfra.

13.5. Specyfikacje kodów dla grupy **printf()**

Wybrane specyfikacje kodu formatu w funkcjach grupy **printf()**.

Kod	Format
c	Gdy brak modyfikatora "l" argument int jest konwertowany w unsigned char i wypisywany jest znak ASCII.
s	Argument typu const char * powinien być wskaźnikiem do napisu. Wypisywany jest napis bez terminatora null-bajtu (\0). Przy podaniu dokładności, wypisywana jest określona liczba znaków.
p	Argumentem powinien być wskaźnik void * , który wypisywany jest szesnastkowo.
n	Liczba wypisywanych dotąd znaków jest zapisywana w argumencie, który musi być wskaźnikiem int * (lub inną odmianą typu całkowitego).
%	Wypisywanie znaku %, czyli kod %%.

13.5. Specyfikacje kodów dla grupy **printf()**

Flagi (znaczniki) formatujące używane w funkcjach grupy **printf()**.
Flagi umieszczane są bezpośrednio po znaku %.

Flaga	Działanie
#	Argument jest przetwarzany w postać alternatywną. Dla formatu o wypisywany jest prefiks 0 , dla formatu x lub X – prefiks 0x lub 0X , a dla formatów: a , A , e , E , f , F , g , G zawsze kropka dziesiętna (nawet gdy nie ma po niej żadnej cyfry). Dla formatów g i G nie są kasowane zera kończące wypisywaną reprezentację liczbową.
0	Dla formatów: d , i , o , x , X , a , A , e , E , f , F , g i G wypisywane są zera wiodące zamiast spacji. W przypadku flagi "-" oraz przy określeniu dokładności reprezentacji formatów: d , i , o , x , X , flaga "0" jest ignorowana.

13.5. Specyfikacje kodów dla grupy **printf()**

Flagi (znaczniki) formatujące używane w funkcjach grupy **printf()**.

Flaga	Działanie
-	Wyrównanie do lewego marginesu (domyślnie mamy do prawego). Nie dotyczy formatu konwersji n – wypełnienie po prawej zerami. Flaga "-" anuluje działanie flagi "0" .
' ' spacja	Dodatni argument w formacie konwersji ze znakiem jest poprzedzany spacją.
+	Dodatni argument jest poprzedzany znakiem '+' (ujemny zawsze ma znak '-'). Flaga ta eliminuje działanie flagi spacji " " .

13.5. Specyfikacje kodów dla grupy **printf()**

Wybrane **modyfikatory** dokładności w funkcjach grupy **printf()**.

Modyfikator	Działanie
hh	Konwersja całkowita (d, i, o, u, x, X) jest konwersją argumentu signed char lub unsigned char . Dla formatu n jest konwersją wskaźnika do signed char .
h	Konwersja całkowita jest konwersją argumentu short int lub unsigned short int . Dla formatu n jest konwersją wskaźnika do short int .
l	Konwersja całkowita jest konwersją argumentu long int lub unsigned long int . Dla formatu n jest konwersją wskaźnika do long int .
ll	Konwersja całkowita jest konwersją argumentu long long int lub unsigned long long int . Dla formatu n jest konwersją wskaźnika do long long int .
L	Modyfikator dla standardu C99. Konwersja zmiennoprzecinkowa (a, A, e, E, f, F, g, G) jest konwersją argumentu long double .

13.5. Specyfikacje kodów dla grupy printf(), p13-5a

Program prezentuje wybrane kombinacje formatów, flag i modyfikatorów.

```
int main ()
{char *string = "Janek ma 'Fischery'.\n";
long l_val = 1212090332; short s_val= 17;
double d_val = 123.137712e5;
printf("Pełny napis: %s\n", string);
printf("Użyto kod (%%.11s): %.11s\n",string);
printf("long int (%%.4ld): %.4ld\n", l_val);
printf("double (%%14.2f): %14.2f\n", d_val);
printf("double (%%-14.2f): %-14.2f\n", d_val);
printf("short int (%%+d): %+d\n", s_val);
printf("short hex (%%4x): %4x\n", s_val);
printf("short hex (%%#4x): %#4x\n", s_val);
printf("short hex (%%04x): %04x\n", s_val);
printf("long w short (%%hx): %hx\n", l_val);
printf("long hex (%%#x): %#x\n", l_val);
printf("long hex (%%x): %x\n", l_val);
system("pause"); return 0; }
```

```
Pełny napis: Janek ma 'Fischery'.
Użyto kod (%%.11s): Janek ma 'F
long int (%%.4ld): 1212090332
double (%%14.2f):      12313771.20
double (%%-14.2f): 12313771.20
short int (%%+d): +17
short hex (%%4x):   11
short hex (%%#4x): 0x11
short hex (%%04x): 0011
long w short (%%hx): 7dc
long hex (%%#x): 0x483f07dc
long hex (%%x): 483f07dc
Press any key to continue . . .
```

13.6. Specyfikacje kodów dla grupy **scanf()**

Wybrane specyfikacje kodu formatu w funkcjach grupy **scanf()**.

Kod	Format
d	Kod wczytuje dziesiętną liczbę całkowitą (bez znaku i ze znakiem). Wskaźnik powinien być wskaźnikiem do typu int .
D	Kod równoważny ld, ignorowany przez niektóre biblioteki.
i	Kod wczytuje dziesiętną liczbę całkowitą (bez znaku i ze znakiem). Wskaźnik powinien być wskaźnikiem do typu int . Wczytywana liczba jest traktowana jako szesnastkowa (prefiks "0x" lub "0X"), oktalna – prefiks "0", dziesiętna – bez prefiksu.
o	Kod wczytuje liczbę całkowitą oktalną bez znaku. Wskaźnik powinien być wskaźnikiem do typu unsigned int .
u	Kod wczytuje dziesiętną liczbę całkowitą bez znaku. Wskaźnik powinien być wskaźnikiem do typu unsigned int .
x, X	Kod wczytuje liczbę całkowitą szesnastkową bez znaku. Wskaźnik powinien być wskaźnikiem do typu unsigned int .

13.6. Specyfikacje kodów dla grupy **scanf()**

Wybrane specyfikacje kodu formatu w funkcjach grupy **scanf()**.

Kod	Format
f, e, g, E	Kod wczytuje liczbę zmiennoprzecinkową (bez znaku i ze znakiem). Wskaźnik powinien być wskaźnikiem do typu float .
s	Kod wczytuje łańcuch znaków różnych od spacji i tabulacji. Wskaźnik do typu char musi wskazywać bufor o rozmiarach do zmieszczenia całej sekwencji znaków i terminatora null-bajtu. Wczytywanie kończy się na znaku niewidocznym (spacja, tabulacja) lub końcu pola.
c	Kod wczytuje łańcuch znaków podanych w długości pola znaków (domyślnie 1 znak). Wskaźnik powinien być wskaźnikiem do typu char i musi wskazywać bufor o stosownych rozmiarach (nie jest dodawany terminator null-bajt). Wiodące spacje nie są pomijane (aby je pominąć musimy jawnie użyć spacji w formacie).
p	Kod wczytuje wartość wskaźnika (odpowiada wpisaniu wskaźnika formatem %p). Wskaźnik powinien być wskaźnikiem do typu void .

13.6. Specyfikacje kodów dla grupy **scanf()**

Wybrane specyfikacje kodu formatu w funkcjach grupy **scanf()**.

Kod	Format
n	Operator wczytywania nie oczekuje niczego na wejściu. Do wskaźnika typu int jest wpisywana liczba znaków dotąd wczytanych z wejścia.
%	Odpowiada literałowi %. Znaki "%%" w łańcuchu formatującym odpowiadają pojedynczemu znakowi "%". Nie dokonują żadnej konwersji i podstawienia.
[]	Kod wczytuje niepustą sekwencję znaków z wyspecjalizowanego w nawiasach zestawu znaków. Wskaźnik do typu char musi wskazywać bufor o rozmiarach do zmieszczenia całego łańcucha znaków i terminator null-bajt. Określenie zestawu znaków w [] musi być zgodne z zasadami definiowania wyrażeń regularnych. Wczytywanie kończy się po napotkaniu pierwszego znaku spoza zestawu lub po dojściu do końca pola ze znakami.

13.6. Specyfikacje kodów dla grupy **scanf()**

Flagi (znaczniki) formatujące używane w funkcjach grupy **scanf()**.

Flaga	Format
*	Anulowanie podstawienia. Konwersja dokonywana jest normalnie. Nie wykorzystuje się żadnego wskaźnika w celu zapisania wyników wczytywania i konwersji. Wynik konwersji jest ignorowany.
a	W standardzie C99 jest równoważne f.
h	Konwersja typu całkowitego: d, i, o, u, x lub typu n . Wskaźnik powinien być wskaźnikiem do typu short int , a nie int .
l	Jeśli mamy konwersję typu całkowitego: d, i, o, u, x , lub typu n , to wskaźnik powinien być wskaźnikiem do typu long int . Dla konwersji zmiennoprzecinkowych: e, f, g używamy wskaźnika typu double . Podwójna flaga " ll " jest równoważna fladze " L ".
L	Jeśli mamy konwersję typu całkowitego: d, i, o, u, x , to wskaźnik powinien być do typu long long . Dla konwersji zmiennoprzecinkowych: e, f, g używamy wskaźnika typu long double .

13.7. Inne operacje plikowe

Wybrane biblioteczne funkcje wejścia-wyjścia.

Funkcja wymusza zapis wszystkich buforowanych danych strumienia wyjściowego **stream**.

int fflush (FILE *stream);

- jeśli argument **stream** jest równy NULL, funkcja opróżnia wszystkie otwarte strumienie wyjściowe, np.:

fflush(stdin);

- nie musi to dotyczyć strumienia jądra systemu operacyjnego, obsługujących np. buforowany zapis na dysku: funkcje **sync()** i **fsync()**.

fflush(stdout);

13.7. Inne operacje plikowe

Wybrane biblioteczne funkcje wejścia-wyjścia.

Usuwanie plików z wnętrza programu:

int remove (const char *pathname);

- funkcja wymaga dołączenia pliku nagłówkowego **stdio.h**,
- argumentem jest nazwa usuwanego pliku (lub ścieżka do niego).

int unlink (const char *pathname);

- funkcja wymaga dołączenia pliku nagłówkowego **unistd.h**,
- argumentem jest nazwa usuwanego pliku (lub ścieżka do niego).

Funkcje usuwają link do pliku (wskazane dowiązanie), a plik zostanie usunięty gdy zostanie zlikwidowane ostatnie dowiązanie.

Materiały zostały opracowane w ramach projektu
„Zintegrowany Program Rozwoju Politechniki Lubelskiej – część druga”,
umowa nr **POWR.03.05.00-00-Z060/18-00**
w ramach Programu Operacyjnego Wiedza Edukacja Rozwój 2014-2020
współfinansowanego ze środków Europejskiego Funduszu Społecznego

NAZWA PRZEDMIOTU
Programowanie strukturalne

Temat wykładu 14.

**Nowe operatory arytmetyczne. Priorytety operatorów.
Zarządzanie pamięcią, data i czas.**

dr hab. inż. Jerzy Montusiewicz, prof. PL

14. Agenda

14.1. Nowe operatory arytmetyczne.

14.2. Podział operatorów.

14.3. Priorytety operatorów.

14.4. Zarządzanie pamięcią.

14.5. Data i czas.

14.6. Wskaźnik stałe i do funkcji.

14.1. Inne operatory arytmetyczne

Skrócony zapis operatora podstawienia w powiązaniu z operatorami arytmetycznymi.

Zamiast

zmienna1 = zmienna1 **op** zmienna2

skrót

zmienna1 **op=** zmienna2

Tak więc mamy: **+=** , **-=** , ***=** , **/=** , np.:

x = x + 10 ➔ x **+=** 10;

y = y * 10 ➔ y ***=** 10

Pamiętaj! W tym połączeniu operator **'='** występuje zawsze na drugim miejscu, po operatorze przypisania.

14.1. Inne operatory arytmetyczne

Operator przecinkowy. Służy do szeregowania wyrażeń i pełni funkcję zbliżoną do spójnika „i”.

np.: $y = 10;$

$x = (y = y - 5, 20/y);$

Ile ostatecznie wynosi x ?

$y = 10 - 5$, więc $y = 5$ następnie $20/5$, a więc $x = 4$.

Kolejność wykonywania działań:

1. Instrukcja w nawiasie przed operatorem przecinkowym.
2. Instrukcja w nawiasie po operatorze przecinkowym.
3. Operacja podstawienia.

Pamiętaj! Operator przecinkowy ma niższy priorytet niż operator podstawienia.

14.1. Inne operatory arytmetyczne

Operator ternarny (trójargumentowy), operator alternatywny.

Wyrażenie_relacyjne ? wyrażenie1 : wyrażenie2;

Gdy **Wyrażenie_relacyjne** zwraca wartość logiczną „prawda” to wykonywane jest **wyrażenie1**, gdy „fałsz” to **wyrażenie2**.

np.:
x = 10;
y = x > 9 ? 100 : 200;

Zapis tradycyjny:
x = 10;
if (x > 9)
y = 100;
else
y = 200;

14.1. Inne operatory arytmetyczne

Ternarny operator alternatywny akceptuje jako argument wyrażenie, także funkcję zwracającą wartość (nie może to być funkcja zwracająca typ void).

np.:

```
wynik = j ? i/j : div_zero()
```

Pamiętaj, że $j \neq 0$

```
int div_zero (void) {  
    printf("dzielenie przez zero jest niemożliwe\n");  
}
```

14.2. Podział operatorów

Podział operatorów występujących w języku C.

1. Ze względu na liczbę argumentów:

- operatory unarne (jednoargumentowe),
- operatory binarne (dwuargumentowe),
- operatory wieloargumentowe.

Operatory unarne to takie gdy występuje tylko jeden argument.

Operatory binarne mają dwa argumenty umieszczone po lewej i prawej stronie znaku/znaków operatora.

Operatory wieloargumentowe mają więcej niż dwa argumenty.

Przykładem jest operator ternarny (trójargumentowy), tzw. alternatywa ternarna '?:' (omówiona w podrozdziale 14.1, slajd 6).

Pamiętaj!

Operatory dwuargumentowe mają łączność lewostronną.

Operatory jednoargumentowe i operator ternarny mają łączność prawostronną.

14.2. Podział operatorów, operatory unarne

Operator	Opis operatorów unarnych	Przykład
!	Negacja bitowa	1 daje 0 , 0 daje 1
~	Negacja bitowa, uzupełnienie do jedynki	~0101 daje 1010
+	Plus jednoargumentowy	+p
-	Minus jednoargumentowy	-p
++	Preinkrementacja lub postinkrementacja	++j , j++
--	Dekrementacja (pre- lub post-)	--k , k--
*	Wskazanie pośrednie (wyłuskanie)	*pointer
&	Adres elementu (ampersand)	&ival
sizeof	Rozmiar zmiennej (tablicy, struktury) lub typu	sizeof(int), sizeof(tab), sizeof ival
(typ)	Jawne rzutowanie typu, zmiana typu	(float)ival, int*)calloc()

14.2. Podział operatorów

2. Ze względu typ operatora:

- operatory arytmetyczne,
- operatory relacyjne,
- operatory logiczne,
- operatory bitowe
- operatory nawiasowe i selekcji.

Operatory arytmetyczne odnoszą się do zmiennych i powodują zmianę wartości zmiennych. Mogą być unarne, binarne i ternarne.

Operatory relacyjne to operatory porównań, których argumentami są wartości liczbowe reprezentowane bezpośrednio lub przez zmienne. Wynikiem tych operatorów jest wartość logiczna prawda lub fałsz (true/false, 1/0).

14.2. Podział operatorów

Operatory logiczne to operatory, których argumentami są wartości logiczne (1/0). Mogą być unarne (!) i binarne. Wynikiem tych operatorów jest wartość logiczna prawda lub fałsz (true/false, 1/0).

Operatory bitowe to operatory, których argumentami są poszczególne bity (1/0) reprezentujące wartość zmiennej. Mogą być unarne (~) lub binarne. Wynikiem tych operatorów są wartości bitów (1/0), które tworzą nową wartość liczbową.

Operatory nawiasowe i selekcji służą do wskazania konkretnego obiektu, elementu, typu, dostępu do pola zmiennych złożonych. Mogą być unarne lub binarne.

14.2. Podział operatorów, operatory arytmetyczne

Operator	Opis operatorów arytmetycznych	Przykład
Operatory multiplikatywne, binarne		
*	Mnożenie	$a*b$, 3.*ival
/	Dzielenie	a/b , 3./ival
%	Dzielenie modulo (reszta z dzielenia)	$k\%2$
Operatory addytywne, binarne		
+	Dodawanie	$c+d$, 3+ival
-	Odejmowanie	$c-d$, 3-ival
Operator warunkowy (ternarny) i wyliczenia		
?:	Wybór jednego z dwóch wyrażeń	$\text{min}=(k<n?k,n)$
,	Operator przecinkowy, obliczenie wartości	$x = (y = y -5, 30/y)$

14.2. Podział operatorów, operatory przypisania

Operator	Opis operatorów przypisania	Przykład
Operatory arytmetyczne		
=	Proste przypisanie	k=9 , k=ival
=	Mnożenie, potem przypisanie	m=3 czyli m=m*3
/=	Dzielenie, potem przypisanie	u/=5
%=	Dzielenie modulo, potem przypisanie	u%=2
+=	Dodawanie, potem przypisanie	u+=3
-=	Odejmowanie, potem przypisanie	u-=3
Operatory bitowe		
&=	Iloczyn bitowy, potem przypisanie	b&=033
^=	Suma modulo 2, potem przypisanie	c^=012
=	Suma bitowa, potem przypisanie	a =4
<<=	Przesunięcie w lewo (w prawo) bitowa,	j<<=2
>>=	potem przypisanie	k>>=1

14.2. Podział operatorów, relacyjne, nawiasowe, selekcji

Operator	Opis operatorów relacyjnych, binarnych	Przykład
<	Mniejszy niż	$u < v$
<=	Mniejszy niż lub równy	$u \leq v$
>	Większy niż	$u > v$
>=	Większy niż lub równy	$u \geq v$
==	Równy	$u == v$
!=	Nierówny (różny)	$u != v$

Operator	Opis operatorów nawiasowych i selekcji	Przykład
()	Zapis strumienia	<code>fflush(stdin)</code>
[]	Odwołanie do elementu tablicy	<code>tab[16]</code> , <code>tab[j]</code>
->	Strzałka, selekcja pola np. struktury	<code>pointer -> pole1</code>
.	Kropka, odwołanie się do pola np. unii	<code>un.ival</code>

14.2. Podział operatorów, operatory logiczne

Operator	Opis operatorów	Przykład
Operatory logiczne, binarne		
&&	Iloczyn logiczny (AND)	$u < v \ \&\& \ k == 5$
	Suma logiczna (OR)	$u <= v \ \ k == 6$
Operatory bitowe, binarne		
&	Bitowy iloczyn logiczny (AND)	$5 \& 3$ daje 1 (0001)
	Bitowa suma logiczna (OR)	$5 3$ daje 7 (0111)
^	Bitowa suma modulo 2 (XOR)	$5 \wedge 3$ daje 6 (0110)
<<	Przesunięcie bitowe w lewo	$8 << 3$ daje 64 $00001000 \rightarrow 01000000$
>>	Przesunięcie bitowe w prawo	$8 >> 3$ daje 1 $00001000 \rightarrow 00000001$

14.3. Priorytet operatorów

Poniżej przedstawiono obowiązujące priorytety wszystkich poznanych operatorów: arytmetycznych, relacyjnych, logicznych, bitowych, nawiasowych i selekcji. Od priorytetu najwyższego do najniższego.

Operatory	Działanie/nazwa
() [] -> .	priorytet, indeks, składowa struktury/unii poprzez wskaźnik, bezpośrednio
! ~ ++ -- (typ) * & sizeof	negacja logiczna, negacja bitowa, inkrementacja, dekrementacja, rzutowanie typu, wartość, adres, rozmiar
* / %	mnożenie, dzielenie modulo
+ -	dodawanie, odejmowanie
<< >>	przesunięcie bitowe w lewo, w prawo
< <= > >=	mniejszy, mniejszy równy, większy, większy równy
== !=	relacja równości?, nierówności?

14.3. Priorytet operatorów

Operatory	Działanie/nazwa
&	iloczyn logiczny bitowy
^	suma modulo 2 bitowa
	suma logiczna bitowa
&&	Iloczyn logiczny
	Suma logiczna
?:	alternatywa ternarna
= += %= -= *= /= &= ^= = <<= >>=	podstawienie i kombinacje podstawienia z operatorami arytmetycznymi i bitowymi
,	przecinek (szeregowanie)

14.4. Zarządzanie pamięcią

Zarządzanie pamięcią w języku C polega na sposobach dynamicznej alokacji obszaru w pamięci wolnej (wykład 9). Prototyp funkcji **malloc()** alokującej **size** bajtów ma postać:

```
void* malloc (size_t size);
```

size_t – alias typu całkowitego bez znaku, zazwyczaj **unsigned long** (zależny od implementacji),

void* – zwraca wskaźnik do początku zarezerwowanego obszaru.

Stąd wynika, że każdorazowo musimy dokonać odpowiedniej konwersji przed przypisaniem tego adresu do zmiennej wskaźnikowej zdefiniowanego typu, np.

```
double * wsk = (double*) malloc (sizeof(double)*size);
```

Do zwolnienia pamięci dynamicznej używamy funkcji **free()**:

```
void free (wsk);
```

14.4. Zarządzanie pamięcią

Zarządca pamięci – odpowiada za przydział i zwalnianie pamięci.

Strategie przydziału pamięci:

- **pierwszy pasujący** – wyszukiwany pierwszy, którego rozmiar jest większy lub równy żądanemu;
- **najlepiej pasujący** - wyszukiwany jest wolny obszar, którego wielkość jest najbliższa żądanej;
- **najgorzej pasujący** - wyszukiwany jest największy wolny obszar.

Powstające problemy:

- **wyciek pamięci** – niezamierzone użycie pamięci przez program, brak zwalnia zaalokowanej wcześniej pamięci ,
- fragmentacja wewnętrzna – programy nie operują na obszarach o dowolnym rozmiarze, lecz blokach danych będących wielokrotnością określonej liczby bajtów,
- fragmentacja zewnętrzna – po wielu operacjach przydzielania i zwalniania bloków pamięci o różnej wielkości, skutek – bloki wolne i zajęte są przemieszane.

14.5. Data i czas

Z poziomu języka C czas i data są określane we współpracy z zegarem czasu rzeczywistego przez system operacyjny.

W systemach z rodziny UNIX początek rachuby czasu (**Epoka**) określa się od dnia 1 stycznia 1970 r. (0 godzin, 0 minut, 0 sekund).

Wartość zliczanych sekund jest przechowywana w zmiennej całkowitej ze znakiem **typu time_t** zdefiniowana w pliku nagłówkowym `time.h`. Typ zmiennej jest w istocie typem **int** (4. bajty ze znakiem) – wypełnienie i przepełnienie w roku 2038.

Funkcja **time()** zwraca wartość o bieżącej wartości z tej zmiennej.

Wyznaczenie daty polega na przeliczeniu czasu wrazonego w sekundach na lata.

Uproszczenia: lata podzielne przez 4 są przestępne (pierwszy wyjątek to 2100 rok), pominięto przestępne sekundy (wynikają z obrotu Ziemi wokół osi i Słońca).

14.5. Data i czas, p14-5a

Obliczenie czasu działania pętli. Test szybkości działania programu ze zmienna umieszczona w rejestrze (register) i pamięci (M.M.Stabrowski).

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

```
#include <time.h>
```

```
#define N 128000
```

Zmienne globalne, poza rejestrem

```
unsigned int i; unsigned int delay;
```

```
int main()
```

```
{ register unsigned int j;
```

```
time_t t;
```

```
printf ("Wyniki dla N = %d\n", N);
```

```
t = time('\0'); Liczba sekund od Epoki
```

```
for (delay=0; delay<N; delay++)
```

```
for (i=0; i<N; i++);
```

```
printf ("Czas bez register: %ld\n", time('\0')-t);
```

```
t = time('\0');
```

```
for (delay=0; delay<N; delay++)
```

```
for (j=0; j<N; j++);
```

```
printf ("Czas z register: %ld\n", time('\0')-t);
```

```
return 0; }
```

```
Wyniki dla N = 64000
Czas bez register: 7
Czas z register: 1
```

```
Wyniki dla N = 128000
Czas bez register: 29
Czas z register: 5
```

```
Wyniki dla N = 128000
Czas bez register: 30
Czas z register: 5
```

14.5. Data i czas, p14-5b

Wyznaczenie bieżącej daty i czasu. Wykorzystanie funkcji z biblioteki **time.h** (M.M.Stabrowski)

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

```
#include <time.h>
```

```
int main()
```

```
{ struct tm {
```

Zawartość struktury tm - czas rozłożony:

```
    int tm_sec;    //sekundy 0-59
```

```
    int tm_min;    //minuty 0-59
```

```
    int tm_hour;    //godziny 0-23
```

```
    int tm_mday;    //dzień miesiąca 1-31
```

```
    int tm_mon;    //miesiąc 1-11
```

```
    int tm_year;    //rok od 1900
```

```
    int tm_wday;    //dzień tygodnia od niedz.
```

```
    int tm_yday;    //dzień roku od 01.01
```

```
    int tm_isday;}; // sezonowa zmiana czasu
```

```
struct tm *ptr; time_t lt;
```

```
lt=time('\0'); ptr=localtime(&lt);
```

```
printf("Po localtime() i asctime(): \n");
```

```
printf(asctime(ptr));
```

```
printf("Tylko po ctime(): \n"); printf(ctime(&lt));
```

```
return 0; }
```

Po localtime() i asctime():

Mon Dec 30 21:00:20 2019

Tylko po ctime():

Mon Dec 30 21:00:20 2019

14.6. Wskaźnik stałe i do funkcji

Stałe wskaźniki:

- wskaźniki na stałą wartość – nie można zmieniać wskazanej wartości

const int *wsk; lub **int const *wsk;**

- stały wskaźnik – nie można przestawić na inny adres

int * const war;

- wskaźnik stały – nie pozwala zmieniać wartości wskazanej zmiennej

const int * const ptr; lub **int const * const ptr;**

Wskaźniki na stałą wartość są przydatne gdy mamy duże obiekty, np. strukturę z kilkoma polami.

14.6. Wskaźnik stałe i do funkcji, p14-6a

Działanie wskaźników stałych różnego rodzaju.

```
int main()
{ int j=0;
  const int *wsk=&j;
  int * const c_wsk=&j;
  int const * const cc_wsk=&j;
  printf("const int *wsk : %#x , %d\n", wsk, *wsk);
  printf("int * const c_wsk : %#x , %d\n", c_wsk, *c_wsk);
  printf("int const * const cc_wsk : %#x , %d\n", cc_wsk, *cc_wsk);
  // *wsk = 1;      // protest kompilatora
  *c_wsk = 100; printf("int * const c_wsk : %#x , %d\n", c_wsk, *c_wsk);
  // *cc_wsk = 3;   // protest kompilatora
  wsk = c_wsk; printf("const int *wsk      : %#x , %d\n", wsk, *wsk);
  // c_wsk = wsk;   // protest kompilatora
  // cc_wsk = wsk;  // protest kompilatora
  return 0; }
```

const int *wsk	:	0x62fe04	,	0
int * const c_wsk	:	0x62fe04	,	0
int const * const cc_wsk	:	0x62fe04	,	0
int * const c_wsk	:	0x62fe04	,	100
const int *wsk	:	0x62fe04	,	100

14.6. Wskaźnik stałe i do funkcji

Wskaźniki do funkcji.

Funkcja ma swoje określone miejsce w pamięci, stąd wynika, że można zdefiniować wskaźnik, który na nią wskaże. Szkielet notacji:

typ (*wsk) (lista_par);

- **typ** – podajemy typ zwaracanej wartości,
- ***wsk** – nazwa wskaźnika,
- **lista-par** – typ i nazwa poszczególnych parametrów oddzielonych przecinkami, lista parametrów może być pusta.

Wskaźnika używamy w ten sam sposób jak funkcję na którą on wskazuje.

Przy wywołaniu funkcji wpisujemy nazwę funkcji, bez użycia nawiasów. Przypisujemy ją do wskaźnika tej funkcji. Kompilator będzie to rozumiał jako adres tej funkcji.

14.6. Wskaźnik stałe i do funkcji, p14-6b

Przykład użycia wskaźnika do funkcji.

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

Prototyp funkcji.

```
float iloraz (int licz, int mian);
```

```
int main()
```

Deklaracja wskaźnika do funkcji.

```
{ float (*wsk_iloraz) (int a, int b);
```

```
    int a=7, b=2;
```

```
    if (b) { wsk_iloraz = iloraz; Wpisujemy nazwę funkcji bez użycia nawiasów.
```

```
        printf("Dla a=%d i b=%d, iloraz = %.2f\n", a ,b, wsk_iloraz(a,b)); }
```

```
    else printf("Nie ma dzielenia przez 0.\n");
```

Wywołanie funkcji przez podanie wskaźnika do funkcji z parametrami.

```
    return 0;
```

```
}
```

```
float iloraz (int licz, int mian)
```

```
{ return (float)licz/mian; }
```

Dla a=7 i b=2, iloraz = 3.50

Gdy b równe zero.

Nie ma dzielenia przez 0.

Materiały zostały opracowane w ramach projektu
„Zintegrowany Program Rozwoju Politechniki Lubelskiej – część druga”,
umowa nr **POWR.03.05.00-00-Z060/18-00**
w ramach Programu Operacyjnego Wiedza Edukacja Rozwój 2014-2020
współfinansowanego ze środków Europejskiego Funduszu Społecznego

NAZWA PRZEDMIOTU
Programowanie strukturalne

Temat wykładu 15.
Preprocesor. Standard C99.

dr hab. inż. Jerzy Montusiewicz, prof. PL

15. Agenda

- 15.1. Preprocesor, przegląd dyrektyw preprocesora.**
- 15.2. Preprocesor, dołączanie – #include.**
- 15.3. Preprocesor, makrodefinicje – #define i #undef.**
- 15.4. Preprocesor, kompilacja warunkowa.**
- 15.5. Standard C99, nowości.**
- 15.6. Standard C99, tablice o zmiennych rozmiarach.**
- 15.7. Standard C99, typ logiczny.**
- 15.8. Standard C99, arytmetyka liczb zespolonych.**
- 15.9. Standard C99, pliki graficzne.**

15.1. Preprocesor, przegląd dyrektyw preprocesora

Preprocesor to wyspecjalizowany edytor tekstowy:

- Edytory tego rodzaju nazywane są makroprocesorami (definiowanie makr: stałe lub funkcje).
- Preprocesor może być używany niezależnie od kompilatora i programów w języku C czy C++.
- Wykonywanie operacji przed wkroczeniem do akcji kompilatora.
- Dyrektywy preprocesora rozpoczynają się znakiem "#" (hash), a ich terminatorem jest znak nowego wiersza (klawisz Enter). (gdy kilka wierszy to znakiem kontynuacji jest "\").

15.1. Preprocesor, przegląd dyrektyw preprocesora

Dyrektywy przetwarzane przez preprocesor:

- dołączanie plików: **#include**,
- definiowanie i usuwanie definicji stałych i funkcji (makr):
#define, #undef,
- kompilacja warunkowa:
#if, #elif, #endif, #else, #ifdef, #ifndef,
- wspomaganie uruchamiania i uzdatniania: **#error, #line**,
- specjalnych opcji kompilacji: **#pragma**.

15.2. Preprocesor, dołączanie – #include

Dyrektywa ta poleca włączyć do tekstu postaci źródłowej programu, który jest przekazywany dalej do kompilatora, wskazany w tej dyrektywie plik. **#include <stdio.h>**

Poszukiwanie w systemowej bibliotece (w katalogu, który zna kompilator).

Czasami pokazujemy podkatalog: **#include <sys/times.h>**

(standard C89 obsługuje 8 poziomów zagnieżdżenia, zaś C99 – 15).

#include "mydefine.h"

Poszukiwanie w bieżącym katalogu.

15.2. Preprocesor, dołączanie – #include, p15-2

Dołączenie własnych bibliotek w katalogu bieżącym.

```
#include "mydefine.h"  
#include "mydefine2.h"  
#include <stdio.h>  
#include <stdlib.h>
```

```
//#include "mydefine2.out"
```

```
//#include "pli2.txt"
```

```
main()
```

```
{ int wiek;  
  wiek = 47;  
  printf (FORMAT, wiek);  
  printf (FORMAT2, wiek);  
  printf (FORMAT3, wiek);  
  exit (1);  
}
```

*Preprocesor podstawił makra
znajdzone w plikach:*

mydefine.h , mydefine2.h

mydefine.h

```
#define FORMAT "Mam %d lat!! \n"
```

```
#define FORMAT2 "Jestem w SREDNIM wieku mam %d lat! \n"
```

mydefine2.h, mydefine2.out, pli2.txt

```
#define FORMAT3 "Jestem mlody mam %d lat. \n"
```

```
Mam 47 lat!!  
Jestem w SREDNIM wieku mam 47 lat!  
Jestem mlody mam 47 lat.
```

15.3. Preprocesor, makrodefinicje – #define, p15-3a

Dyrektywa **#define** służy do definiowania stałych (parametrów) i makr, czyli funkcji o kodzie **wstawianym** w tekst programu (a nie wywoływanych, czyli skok i powrót).

Używanie **WERSALIKÓW** jako nazw makr ułatwia czytanie programu i odszukanie miejsc w których mają miejsce **makropodstawienia**.

Podstawienia MAKRA w funkcji printf(), argument a nie stała znakowa.

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

```
#define VERSION "version 10.1\n"
```

```
#define DLUGI TEKST "Ten tekst jest tak długi, że nie mieści się \n\nw jednej linii dyrektywy.\n"
```

Znak "\n" kontynuuje zapis w kolejnej linii

```
main()
```

```
{printf ("OPERATING SYSTEM VERSION\n");
```

```
printf (DLUGI TEKST);
```

```
printf (VERSION);
```

```
return 0;
```

```
}
```

```
OPERATING SYSTEM VERSION
Ten tekst jest tak długi, że nie mieści się w jednej linii dyrektywy.
-- version 11.1
```


15.3. Preprocesor, makrodefinicje – #define, p10-3b1, b2, b3

Makra typu funkcyjnego mogą zawierać w swoich definicjach argumenty (tak jak funkcje). **MAKRA typu funkcyjnego.**

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

```
#define MIN(a,b) ((a)<(b)? (a):(b))
```

```
main()
```

```
{ int x, y;
```

```
printf("wprowadz dwie wartosci (int,int): ");
```

```
scanf("%d,%d",&x,&y);
```

```
printf ("Minimalna wartoscia jest %d\n", MIN((x),(y)));
```

```
return 0; }
```

To jest po podstawieniu

```
printf ("Minimalna wartoscia jest %d\n", ((x)<(y)? (x):(y)));
```

Pamiętaj! W **MAKRACH** brak sprawdzenia typów zmiennych.

Znaczenie nawiasów:

```
#define F1(y) y+2.718
```

```
#define F2(y) (y+2.718)
```

→

```
Wartosc F1 bez nawiasow = 6.718
```

```
Wartosc F2 z nawiasami = 9.436
```

Operator ternarny

*Stosujemy dodatkowe nawiasy
zobacz (a):(b)*

```
wprowadz dwie wartosci (int,int): 9,2  
Minimalna wartoscia jest 2
```

15.3. Preprocesor, makrodefinicje – #define i #undef

MAKRO do konstruowania pętli.

```
#define DOBY(i,from,to,by) for(i=(from)-1;i<(to);i+=(by))
```

MAKRO do konwersji tablicy dwuwymiarowej w tablicę jednowymiarową.

```
#define INDEXC(i,j,ndim) [(j)+(i)*(ndim)]
```

Do odwoływania definicji MAKRO służy dyrektywa **#undef**.

15.4. Preprocesor, kompilacja warunkowa

Kompilacja warunkowa umożliwia pomijanie lub uwzględnianie sekwencji kodu źródłowego programu. Służą do tego dyrektywy kompilacji:

#if, #elif, #endif, #else, #ifdef, #ifndef

Dyrektywy te mogą realizować operacje podobne jak instrukcje **if** czy **if-else** języka C. Dyrektywy te znajdują się wewnątrz ciała funkcji. Różnica polega na tym, że następuje modyfikacja tekstu programu przekazywanego kompilatorowi.

Ogólna postać dyrektywy **#if**:

#if wyrażenie_relacyjne

instrukcja / je

#endif

Jeśli **wyrażenie_relacyjne** ma wartość logiczną „prawda”, to **instrukcja / je** znajdzie się w kodzie źródłowym przekazanego do kompilatora.

Ważne aby wartość tą można było określić na etapie kompilacji.

15.4. Preprocesor, kompilacja warunkowa

Dyrektywa **#elif** i **#else** współpracują z dyrektywą **#if**.

```
#if   wyrażenie_relacyjne1
      instrukcje_1;
#elif wyrażenie_relacyjne2
      instrukcje_2;
#else
      instrukcje_3;
#endif
```

Jeśli **wyrażenie_relacyjne1 / 2** mają wartość „prawda” to **instrukcje_1** albo **instrukcje_2** (itd. ...), w przeciwnym razie (**#else**) **instrukcje_3**.

15.4. Preprocesor, kompilacja warunkowa, p15-4a

Definiowanie różnych typów zmiennych. Dyrektywy kompilacji warunkowych.

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

```
#define SIZE 30
```

```
main()
```

```
{ int ix;
```

*Kod programu zależny
od wartości **SIZE***

```
  #if SIZE<=50
```

```
    float tablica[SIZE]; printf("SIZE <= 50\n-Wersja float \n");
```

```
  #elif SIZE<100
```

```
    double tablica[SIZE]; printf("SIZE <= 100\n--- Wersja double \n");
```

```
  #else
```

```
    double tablica[0]; printf("Problem zbyt duży – wyskok!\n"); return -1;
```

```
  #endif
```

```
    for (ix=0; ix<SIZE; ix++)
```

```
        tablica[ix] = 1.0 + ix;
```

*Musi pojawić się deklaracja tablicy
bo kompilator zawsze przetwarza*

```
    printf ("Wypełnienie tablicy \n") ;
```

zapis pętli for.

```
    printf ("25-ty element = %f\n",tablica[24]);
```

```
    return 0;
```

```
}
```

```
SIZE <= 50
-Wersja float
Wypełnienie tablicy
25-ty element = 25.00
```

```
SIZE <= 100
--- Wersja double
Wypełnienie tablicy
25-ty element = 25.00
```

15.4. Preprocesor, kompilacja warunkowa

Dyrektywy **#ifdef** i **#ifndef** kontrolują istnienie lub nieistnienie definicji makra (stałej określonej w dyrektywie **#define**).

```
#ifdef makro (stala)  
    instrukcja / je;  
#endif
```

W postaci źródłowej przekazywanej kompilatorowi pojawia się **instrukcja / je**, jeśli zdefiniowane jest MAKRO, niezależnie od jego wartości.

Dyrektywa **#ifndef** przekazuje kompilatorowi instrukcje ze swego zasięgu, gdy makro nie jest zdefiniowane.

Dyrektywa **#ifdef** ma równoważną postać **#if defined**, zaś **#ifndef** równoważną postać **#if !defined**.

W przypadku dyrektyw kompilacji warunkowej standard C89 wymaga aby kompilator obsługiwał 8 poziomów zagnieżdżenia dyrektyw (C99 oczekuje obsługi 63 poziomów zagnieżdżenia).

15.4. Preprocesor, kompilacja warunkowa, p15-4b

Dyrektywy kompilacji warunkowych, przykład od M. Stabrowski.

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

```
#define SIZE 500
```

*Istnieje definicja makro **TYPE***

```
#define TYPE
```

```
#ifndef TYPE
```

*Warunkowo zdefiniowano inne
makra **REAL** oraz **VERSION***

```
    #define REAL double
```

```
    #define VERSION "double"
```

```
#else
```

```
    #define REAL float
```

```
    #define VERSION "float"
```

```
#endif
```

```
int main()
```

```
{ int ix;
```

```
    REAL values[SIZE];
```

```
    printf("%s version\n", VERSION);
```

```
    for (ix=0; ix<SIZE; ix++) values[ix] = 1.0 + ix;
```

```
    printf ("array 'values' filled\n");    return 0;
```

```
}
```

```
double  
array 'values' filled
```

```
float  
array 'values' filled
```

*Definicja makra **TYPE**
skomentowana*

15.5. Standard C99, nowości

Nowe słowa kluczowe:

- **_Bool** – do obsługi nowego typu zmiennych logicznych.
- **_Complex**, **_imaginary** – obsługują zmienne typu zespolonego.

Dodano:

- tablice o zmiennych rozmiarach,
- arytmetyka liczb zespolonych,
- typ long long,
- Komentarze jednowierszowe `/**` (można je zagnieżdżać w `/* ... */`,
- **inline** – potraktowanie kodu funkcji jako kodu wplatanego, co zwiększa rozmiar kodu, ale przyspiesza wykonanie.
- **restrict** – ogranicza dostęp do pamięci za pomocą wskaźnika i ułatwia eliminację części błędów użycia wskaźników.

Wycofano:

- usunięcie domniemanego typu **int** w deklaracji funkcji,
- usunięcie domniemanej deklaracji funkcji – wymusza to do stosowania prototypów funkcji.

15.5. Standard C99, nowa postać instrukcji return

W C89 instrukcja **return** służy do powrotu z funkcji w miejsce jej wywołania i do zwrócenia jakiejś wartości (czasami nic nie zwraca – wtedy występuje bez argumentu).

W C99 dla funkcji różnych od typu **void**, **return** zawsze zwraca jakąś wartość nawet gdy chodzi tylko o powrót.

15.5. Standard C99, nowa postać instrukcji return, p15-5a

Różne rodzaje wyjścia z funkcji – return.

```
int expon (int podst, int wyk);
```

```
int main()
```

```
{ expon (10, 3); expon (-3, -4);  
  expon (10, -3); expon (2, 10);  
  return 0; }
```

Obliczanie całkowitej dodatniej potęgi i jej wydruk

```
int expon (int podst, int wyk)
```

```
{ int temp;
```

```
if(wyk < 0) { printf (" wykladnik = %d ujemny!\n", wyk);
```

```
return;      Gdy wykładnik jest ujemny – return bez argumentu, nic nie zwraca.
```

```
temp=1;
```

W C99 instrukcja powinna cos zwracać, np. return 0

```
printf (" podstawa = %2d; wykladnik = %2d; ", podst, wyk);
```

```
for ( ; wyk; wyk--) temp = podst*temp;
```

```
printf ("POTEGA=%d\n", temp);
```

```
return temp;
```

```
}
```

```
podstawa = 10; wykladnik = 3;   POTEGA=1000  
wykladnik = -4 ujemny!  
wykladnik = -3 ujemny!  
podstawa = 2; wykladnik = 10;  POTEGA=1024
```

15.5. Standard C99, nowe konstrukcje, p15-5b

Brak prototypu, definicje funkcji należy umieścić przed jej pierwszym użyciem. Funkcja zwracająca inny typ niż "int".

`#include <stdio.h>`

*Najpierw funkcja **dblmul()***

double dblmul(double x, double y)

```
{ printf("-- wewnątrz funkcji: %s\n", __func__);  
  double temp; // Obliczanie iloczynu.  
  temp = x * y;  
  printf ("czynniki = %.2lf i %.2lf; ILOCZYN = %.2lf\n", x, y, temp);  
  return temp; }
```

Podwójny ukośnik, może zajmować część wiersza,

int main() *Funkcja **main()** na końcu*

```
{ printf("++ wewnątrz funkcji: %s\n", __func__);  
  double jeden, dwa, produkt;  
  jeden = 1.1; dwa = 2.2;  
  produkt = dblmul(jeden, dwa);  
  printf("++ wewnątrz funkcji: %s\n", __func__);  
  printf ("czynniki = %.2lf i %.2lf; ILOCZYN = %.2lf\n", jeden, dwa, produkt);  
  printf("++ wewnątrz funkcji: %s\n", __func__);  
  jeden = 1.77; dwa = 17.88;  
  produkt = dblmul(jeden, dwa);  
  printf("++ wewnątrz funkcji: %s\n", __func__);  
  printf ("czynniki = %.2lf i %.2lf; ILOCZYN = %.2lf\n", jeden, dwa, produkt);  
  return 0; }
```

```
++ wewnątrz funkcji: main  
-- wewnątrz funkcji: dblmul  
czynniki = 1.10 i 2.20; ILOCZYN = 2.42  
++ wewnątrz funkcji: main  
-- wewnątrz funkcji: dblmul  
czynniki = 1.77 i 17.88; ILOCZYN = 31.65  
++ wewnątrz funkcji: main
```

15.5. Standard C99, nowe konstrukcje, p15-5c

Predefiniowane identyfikatory:

- `__func__` – przekazuje informację w jakiej funkcji się znajduje,
- `__DATE__` – data w chwili kompilacji,
- `__TIME__` – godzina w chwili kompilacji,
- `__FILE__` – łańcuch, który zawiera nazwę pliku kompilowanego,
- `__LINE__` – definiuje numer liniiki.

int main()

```
{ printf(" w funk.: %s\n", __func__);  
  printf(" data: %s\n", __DATE__);  
  printf(" czas: %s\n", __TIME__);  
  printf(" linia: %d\n", __LINE__);  
  printf(" file: %s\n", __FILE__);  
  return 0;  
}
```

```
w funk.: main  
data: Dec 21 2019  
czas: 00:57:42  
linia: 13  
file: F:\dydaktyka\2019_projekt_nowa  
informatyka\0_Programowanie strukturaln  
e_wyk|ady-1\programy do wyk|ad~w\W_15\p  
15.5c.c
```

15.6. Standard C99, tablice o zmiennych rozmiarach, p15-6a

Tablice lokalne o zmiennych rozmiarach. Rozmiar tablicy jest konkretyzowany w fazie wykonywania programu. Tablica nie jest w pełni dynamiczna.

```
void FillArray (int n_rows, int n_cols);
```

```
int main()
```

```
{ int NoRows, NoCols;  
  printf("Array of rows & cols : ");  
  scanf("%d %d", &NoRows, &NoCols);  
  FillArray (NoRows, NoCols);  
  return 0; }
```

*Funkcja z tablicą lokalną o zmiennych rozmiarach **Array(NoRows, NoCols)**.*

← Wymiary sparametryzowane.

```
void FillArray (int n_rows, int n_cols)
```

```
{ float Array_1[n_rows][n_cols]; Deklaracja lokalnej tablicy we wnętrzu funkcji.
```

```
  int iR, iC;
```

```
  for (iR=0; iR<n_rows; iR++){ Operator sizeof działa również w fazie wykonania
```

```
    for (iC=0; iC<n_cols; iC++) Array_1[iR][iC] = iR+1.0 + iC*0.1;}
```

```
  printf ("Array size = %d\n", sizeof(Array_1));
```

```
  for (iR=0; iR<n_rows; iR++)
```

```
  { for (iC=0; iC<n_cols; iC++)
```

```
    printf ("%3.1lf ", Array_1[iR][iC]);
```

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

Sprawdzenie zawartości tablicy.

```
Array of rows & cols : 3 5  
Array size = 60  
1.0  1.1  1.2  1.3  1.4  
2.0  2.1  2.2  2.3  2.4  
3.0  3.1  3.2  3.3  3.4
```


15.6. Standard C99, tablice o zmiennych rozmiarach

Tablice o zmiennych rozmiarach są odmianą tablic służącą **do prawdziwego dynamicznego definiowania tablic** (najczęściej) zlokalizowanych we wnętrzu struktur.

Zawartość elastycznej struktury:

- **przynajmniej jeden element** składowy zadeklarowany w sposób konwencjonalny,
- **ostatni zadeklarowany element** powinien być tablicą bez podanego rozmiaru (puste nawiasy kwadratowe).

Należy **zastosować dynamiczną alokację tablicy w pamięci** (uwzględniając również inne nieelastyczne elementy struktury)

Dostęp do elementów struktury odbywa się **przez zastosowanie wskaźników**.

15.6. Standard C99, tablice o zmiennych rozmiarach, p15-6b

Elastyczne tablice w strukturach. Wypełnianie, operator ->.

```
#include <stdio.h>
#include <stdlib.h>
#define FLEX_SIZE 6
int main()
{ int k, l;
  typedef struct
  { int one, two;
    float FlexArr[];
  } Elastic_Struct;
  Elastic_Struct *Pointer;
  Pointer = (Elastic_Struct *)malloc(sizeof(Elastic_Struct) + F_SIZE*sizeof(float));
  // Wypelnienie tablicy elastycznej
  for (k=0; k<F_SIZE; k++)
    Pointer->FlexArr[k] = (k+1) * 1.1;
  for (l=0; l<F_SIZE; l++)
    printf ("%4.2f ", Pointer->FlexArr[l]);
  printf ("\n");
  return 0;
}
```

typedef do zmiany definicji nazwy struktury

Elastyczne elementy struktury na końcu definicji struktury

Deklaracja wskaźnika do struktury typu Elastic_Struct

Alokacja pamięci - jawne rzutowanie na typ strukturalny

Sprawdzenie zawartości

1.10	2.20	3.30	4.40	5.50	6.60
------	------	------	------	------	------

15.7. Standard C99, typ logiczny `_Bool`, p15-7

Nowy typ danych do przechowywania wartości logicznych (stałe) **true** (=1) oraz **false** (=0). Należy dołączyć plik nagłówkowy **stdbool.h**

Te wartości logiczne to w istocie stałe liczbowe **0** oraz **1**, a nazwy (makra) **false** i **true** są nazwami stałych liczbowych.

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

```
#include <stdbool.h>
```

```
int main(void)
```

```
{ _Bool a1, b1;    Typ logiczny: bool albo _Bool
```

```
    int new_int;
```

```
    a1 = true;
```

```
    b1 = false;
```

```
    a1 = -99;
```

```
    printf ("a1 = %d; b1 = %d\n", a1, b1);
```

```
    new_int = 7;
```

```
    printf ("7 to logiczne: %d\n", (bool)new_int);
```

```
    new_int = 0;
```

```
    printf ("0 to logiczne: %d\n", (_Bool)new_int);
```

```
    return 0; }
```

```
a1 = 1; b1 = 0
7 to logiczne: 1
0 to logiczne: 0
```

*Jawne rzutowanie
na typ logiczny*

15.8. Standard C99, arytmetyka liczb zespolonych: `_Complex`

Typ zespolony wprowadzono ze względu na zastosowanie w metodach numerycznych.

Jako **typ zespolony** można deklarować **zmienne typu float, double oraz long double**. Słowem kluczowym jest **`_Complex`**. Jednostka urojona implementowana jest za pomocą makra **`_Complex_I`** lub jako **`I`**.

- **`creal()`** – funkcja wyznaczająca wartość absolutną części rzeczywistej.
- **`cimag()`** – funkcja wyznaczająca wartość absolutną części urojonej,
- **`cabs()`** – funkcja wyznaczająca wartość absolutną liczby zespolonej.
- istnieją również funkcje do wyznaczania modułu i argumentu wartości zespolonej.

Należy dołączyć plik nagłówkowy **`complex.h`**

Przy kompilacji czasami należy wskazać bibliotekę matematyczną **`lm`**

15.8. Standard C99, liczby i zmienne zespolone, p15-8

Testowanie makr complex i imaginary.

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

```
#include <complex.h>
```

```
int main(void)
```

```
{ double _Complex a1, a2, a3;
```

```
double module;
```

```
a1 = 2.33+_Complex_I*7.22;
```

```
a2 = 5.11+I*3.22;
```

```
printf ("a1 real    = %4.2f\n", creal(a1));
```

```
printf ("a1 imaginary = %4.2f\n", cimag(a1));
```

```
printf ("a2 = %4.2f %4.2f\n", a2, a2); Nieprawidłowy zapis.
```

```
printf ("a1 = %4.2f %5.2f; a2 = %4.2f %4.2f\n", creal(a1), cimag(a1), \
```

```
creal(a2), cimag(a2));
```

```
a3 = a1 + I*cimag(a2);
```

```
printf ("a3 = %4.2f %4.2f\n", creal(a3), cimag(a3));
```

```
module = cabs(a3);
```

```
printf ("Modul liczby a3= %4.2f\n", module);
```

```
printf("Rozmiar liczby zespolonej: %d \n", sizeof(module));
```

```
return 0; }
```

```
a1 real      = 2.33
a1 imaginary = 7.22
a2 = 0.00 0.00
a1 = 2.33 7.22; a2 = 5.11 3.22
a3 = 2.33 10.44
Modul liczby a3 = 10.70
Rozmiar liczby zespolonej: 8
```

15.9. Standard C99, pliki graficzne, p15-9

Pliki graficzne, M.M.Stabrowski.

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

```
int main()
```

```
{ const int dimx = 400;
```

```
  const int dimy = 400;
```

```
  int i, j;
```

```
  FILE * fp = fopen("first.ppm", "wb");
```

```
  fprintf(fp, "P6\n%d %d\n255\n", dimx, dimy);
```

```
  for(j=0; j<dimy; ++j)
```

```
    {for(i=0; i<dimx; ++i)
```

```
      {static unsigned char color[3];
```

```
        color[0]=i % 255; //red
```

```
        color[1]=j % 255; //green
```

```
        color[2]=i*j % 255; //blue
```

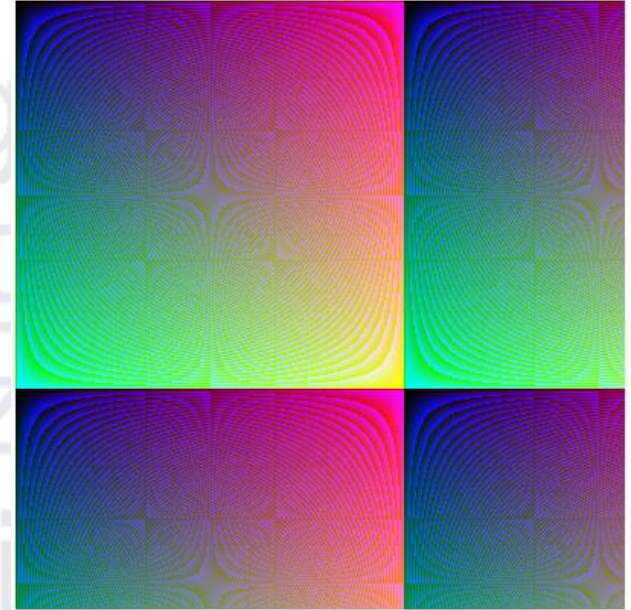
```
        fwrite(color,1,3,fp); }
```

```
      fclose(fp);
```

```
    }
```

```
  return 0;
```

```
}
```



Materiały zostały opracowane w ramach projektu
„Zintegrowany Program Rozwoju Politechniki Lubelskiej – część druga”,
umowa nr **POWR.03.05.00-00-Z060/18-00**
w ramach Programu Operacyjnego Wiedza Edukacja Rozwój 2014-2020
współfinansowanego ze środków Europejskiego Funduszu Społecznego