# Podstawy programowania w C

Witold Paluszyński
witoldp@pwr.wroc.pl
http://sequoia.ict.pwr.wroc.pl/~witold/



Niniejszy dokument zawiera materiały do wykładu na temat podstawowych elementów programowania w języku C. Jest on udostępniony pod warunkiem wykorzystania wyłącznie do własnych, prywatnych potrzeb i może być kopiowany wyłącznie w całości, razem z niniejszą stroną tytułową.

# Przykładowy program w C

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

void main() {
  float a, b, c, aa, d, sqrtd;

  printf("Program wylicza rozwiazania dwumianu kwadratowego.\n");
  printf("Podaj wspolczynnik a:\n");    scanf("%f",&a);
  printf("Podaj wspolczynnik b:\n");    scanf("%f",&b);
  printf("Podaj wspolczynnik c:\n");    scanf("%f",&c);

  aa = 2.0 * a;
  d  = (b*b) - (4.0*a*c);

  if (a != 0.0) {
    if (d == 0.0)
      printf("Istnieje jedno rozwiazanie: %f\n", -b/aa);
    else if  (d > 0.0) {
      sqrtd = (float) sqrt( (double)d );
      printf("Istnieja dwa rozwiazania rzeczywiste:\n");
      printf("  x1 = %f\n", (-b - sqrtd) / aa);
      printf("  x2 = %f\n", (-b + sqrtd) / aa);
```

```
    }
    else { /* czyli d <= 0 */
      sqrtd = (float) sqrt( (double)-d );
      printf("Istnieja dwa rozwiazania zespolone:\n");
      printf("  x1 = %f + %f i\n", -b/aa, sqrtd/aa);
      printf("  x2 = %f - %f i\n", -b/aa, sqrtd/aa);
    }
  }
  else { /* czyli a jest 0 */
    if (c == 0.0)
      printf("Dwumian jest jednomianem, jedyne rozwiazanie: x = 0.0\n");
    else if (b == 0.0) /* ale wiemy a=0 i c!=0 */
      printf("Dwumian sprzeczny (%f = 0.0), blad w danych.\n",c);
    else /* czyli b!=0 i c!=0 */
      printf("Dwumian jest jednomianem, jedno rozwiazanie: %f\n",-c/b);
  }
}
```

# Biblioteka wejścia/wyjścia `stdio` (wstęp): funkcje `getchar` i `putchar`

Przykłady z podręcznika Kernighana i Ritchie: „Język ANSI C” — programy kopiujące znaki z wejścia na wyjście:

```
#include <stdio.h>

/* copy input to output; 1st version */
main()
{
  int c;

  c = getchar();
  while (c != EOF) {
    putchar(c);
    c = getchar();
  }
}
```

```
#include <stdio.h>

/* copy input to output; 2nd version */
main()
{
  int c;

  while ((c = getchar()) != EOF)
    putchar(c);
}
```

# Kolejne przykłady z podręcznika K&R — programy zliczające znaki z wejścia

```
#include <stdio.h>

/* count characters in input; 1st version */
main()
{
  long nc;

  nc = 0;
  while (getchar() != EOF)
    ++nc;
  printf("%ld\n", nc);
}
```

```
#include <stdio.h>

/* count characters in input; 2nd version */
main()
{
  double nc;

  for (nc = 0; getchar() != EOF; ++nc)
    ;
  printf("%.0f\n", nc);
}
```

# Dalsze przykłady z K&R — zliczanie wierszy i wyrazów

```
#include <stdio.h>

/* count lines in input */
main()
{
  int c, nl;

  nl = 0;
  while ((c = getchar()) != EOF)
    if (c == '\n')
      ++nl;
  printf("%d\n", nl);
}
```

```
#include <stdio.h>

#define IN 1     /* inside a word */
#define OUT 0    /* inside a word */

/* count lines, words, and characters in input */
main()
{
  int c, nl, nw, nc, state;

  state = OUT;
  nl = nw = nc = 0;
  while ((c = getchar()) != EOF) {
    ++nc;
    if (c == '\n')
      ++nl;
    if (c == ' ' || c == '\n' || c == '\t')
      state = OUT;
    else if (state == OUT) {
      state = IN;
      ++nw;
    }
  }
  printf("%d %d %d\n", nl, nw, nc);
}
```

# Inny przykład z K&R — zliczanie cyfr, użycie tablic

```
#include <stdio.h>

/* count digits, white space, others */
main()
{
  int c, i, nwhite, nother;
  int ndigit[10];

  nwhite = nother = 0;
  for (i = 0; i < 10; ++i)
    ndigit[i] = 0;

  while ((c = getchar()) != EOF)
    if (c >= '0' && c <= '9')
      ++ndigit[c-'0'];
    else if (c == ' ' || c == '\n' || c == '\t')
      ++nwhite;
    else
      ++nother;

  printf("digits =");
  for (i = 0; i < 10; ++i)
    printf(" %d", ndigit[i]);
  printf(", white space = %d, other = %d\n",
         nwhite, nother);
}
```

# Użycie funkcji w programach w C

```
#include <stdio.h>

float fun1(float x);          /* prototyp funkcji */
void  fun2(int, int);         /* inny prototyp */

int   fun3(int i) {           /* deklaracja funkcji bez prototypu */
  return (i=='\n' || i=='\t' || i==' ');
}

void main() {
  int i,j,k;
  float f;

  while ((i = getchar()) != EOF)
    if (fun3(i)) {
      ...
      fun2(j,k);
      f = fun1(f);
    }
  exit(0);                    /* "poprawny" kod wyjscia */
}

float fun2(int a, int b) {    /* deklaracja funkcji z wczesniejszym prototypem */
 ...
}
```

# Użycie modułów programowych w C

Pliki nagłówkowe (ang. *header files*) modułów programowych (ale nie modułu programu głównego) zawierają prototypy funkcji, i inne zewnętrzne deklaracje: zmiennych, stałych, typów danych, itp.

`fun.h` (plik nagłówkowy modułu funkcji):

```
#include <stdio.h>            /* moze byc potrzebne w deklaracjach */

int   fun1(int i);                    /* tylko prototypy ... */
void  fun2(int, int);                 /* funkcji eksportowanych */
```

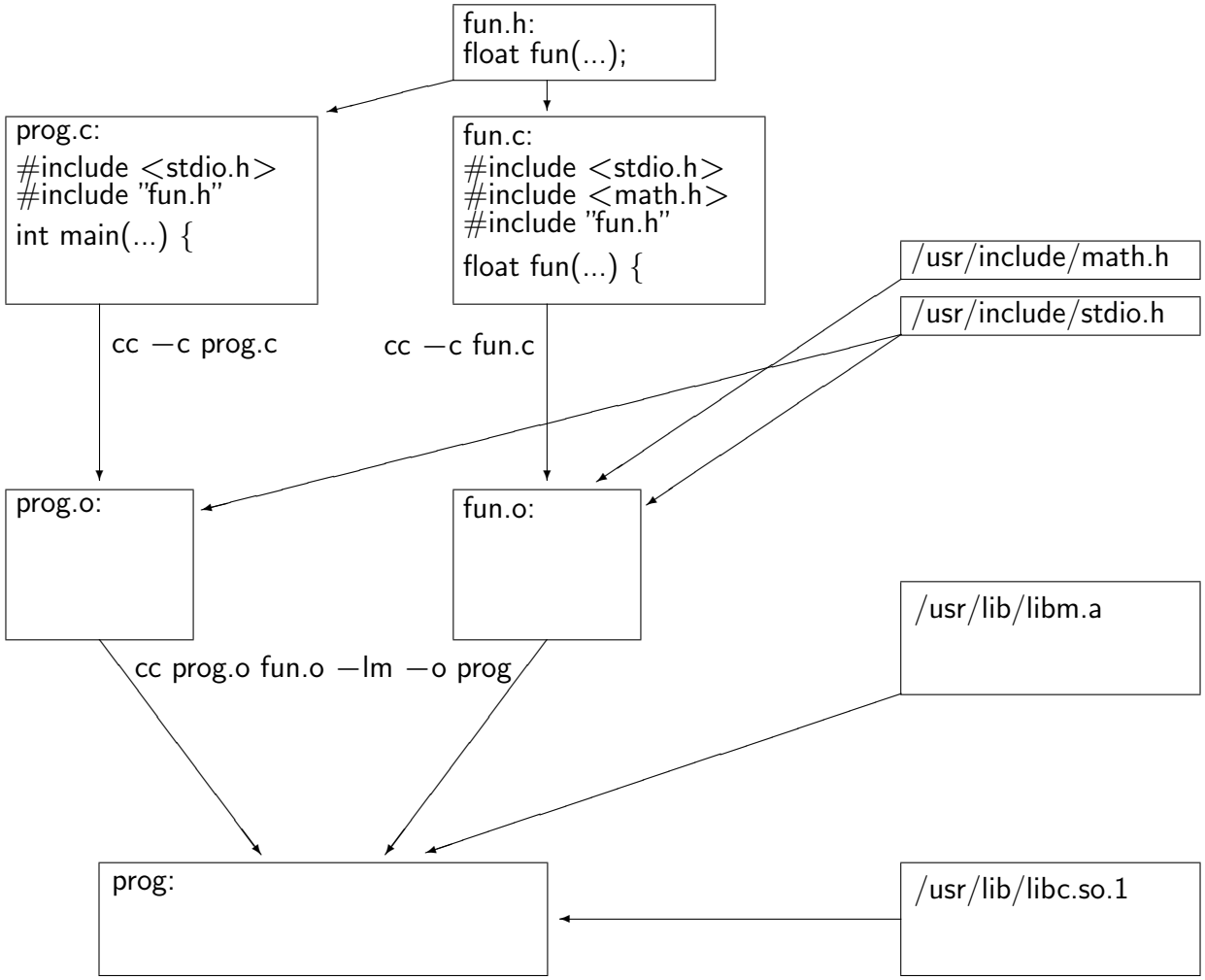
`fun.c` (plik źródłowy modułu funkcji):

```
#include "fun.h"              /* wczytuje tresc pliku naglowkowego */

int   fun1(int i) { ... }             /* funkcja eksportowana */
void  fun2(int a, int b) { ... }      /* inna funkcja eksportowana */
float fun3(float x) { ... }           /* funkcja wewnetrzna modulu */
```

`prog.c` (program główny):

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

void main() { ... if (fun1(i)) { ... fun2(j,k); exit(0); } ... }
```

# Rozdzielna kompilacja programów




wywołanie pełnej kompilacji:

```
cc prog.c fun.c -lm -o prog
```

```
cc -c prog.c
cc -c fun.c
cc prog.o fun.o -lm -o prog
```

# Kompilacja programu — narzędzie make

skrypt do kompilacji programu:

```
cc -c prog.c
cc -c fun.c
cc prog.o fun.o -lm -o prog
```

specyfikacja Makefile:

```
prog: prog.o fun.o
        cc prog.o fun.o -lm -o prog
prog.o: prog.c fun.h
        cc -c prog.c
fun.o: fun.c fun.h
        cc -c fun.c
```

# Opcje wywołania kompilatora C (wspólne)

Tradycyjnie, kompilatory C rozpoznają te opcje jednakowo:

**-onazwa** umieść postać wynikową kompilacji w pliku *nazwa*, domyślnie *a.out* dla postaci programu wykonywalnego, *nazwazrodla.s* dla postaci asemblerowej, i *nazwazrodla.o* dla postaci binarnej

**-c** pomiń ostatnią fazę kompilacji (linker), nie twórz programu wynikowego, pozostaw postać binarną .o

**-g** wpisz w program binarny dodatkowe informacje dla debuggera

**-lbib** powoduje przeglądanie przez linker biblioteki *bib*, w pliku o nazwie `libbib.a` lub `libbib.so` w kartotece `/usr/lib` lub w innych zdefiniowanych ścieżką linkera

**-S** wykonaj tylko pierwszą fazę kompilacji do kodu asemblera .s

**-On** wykonaj optymalizację kodu poziomu *n* (domyślnie poziom 2, który jest na ogół bezpieczny)

**-w** pomiń ostrzeżenia (opcja zwykle szkodliwa)

# Opcje wywołania kompilatorów (różne)

Niestety, niektóre ważne i pożyteczne opcje występują tylko dla niektórych kompilatorów, lub mają inną postać:

-**V** wyświetlaj wywołania kolejnych faz kompilacji (Sun cc)

-**v** wyświetlaj wywołania kolejnych faz kompilacji (HP cc, GNU gcc)

-**Xc** ścisłe przestrzeganie standardu ANSI C (Sun cc)

-**Aa** ścisłe przestrzeganie standardu ANSI C (HP cc)

-**ansi** przestrzeganie standardu ANSI C (GNU gcc)

-**pedantic** ścisłe przestrzeganie standardu ANSI C (GNU gcc)

-**Wall** wyświetlanie ostrzeżeń o wszelkich „dziwnych" konstrukcjach programowych (GNU gcc)

# Przykład `wiz`: wersja początkowa

Przykład z książki Kernighana i Pike'a „Unix Programming Environment" — wizualizacja znaków binarnych:[1] `cat plik | wiz`

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

main()
{
  int c;

  while ((c = getchar()) != EOF)
    if (isascii(c) &&
        (isprint(c) || c=='\n' || c=='\t' || c==' '))
      putchar(c);
    else
      printf("\\%03o", c);
  exit(0);
}
```

[1] Przykład został trochę zmieniony na potrzeby tego kursu.

# Przykład `wiz`: argumenty wywołania programu

Druga wersja programu — opcjonalne usuwanie znaków binarnych sterowane argumentem wywołania programu:

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

int main(int argc, char *argv[]) {
  int c, strip = 0;

  if (argc > 1 && strcmp(argv[1], "-s") == 0)
    strip = 1;

  while ((c = getchar()) != EOF)
    if (isascii(c) &&
        (isprint(c) ||  c=='\n' || c=='\t' || c==' '))
      putchar(c);
    else if (!strip)
      printf( "\\%03o", c);
  return 0;
}
```

Argumenty opcjonalne (dopuszczalne ale nieobowiązkowe) zwyczajowo oznacza się wybranymi literami poprzedzonymi znakiem minusa.

# Przykład `wiz`: trzecia wersja — użycie funkcji

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

int strip = 0;      /* zm.globalna: 1 => usuwanie znakow specjalnych */
void wiz();         /* prototyp funkcji wizualizacji calego pliku stdin */

int main(int argc, char *argv[])
{
  if (argc > 1 && strcmp(argv[1], "-s") == 0)
    strip = 1;

  wiz();
  return 0;
}

void wiz()
{
  int c;

  while ((c = getchar()) != EOF)
    if (isascii(c) &&
        (isprint(c) || c=='\n' || c=='\t' || c==' '))
      putchar(c);
    else if (!strip)
      printf("\\%03o", c);
}
```

# Przykład `wiz`: czwarta wersja — moduł programowy

wiz.h:

```
extern int strip; /*zm.glob.*/
void wiz();       /*prototyp*/
```

main.c

```
#include "wiz.h"

int strip = 0;    /*zm.glob.*/

int main(int argc,char *argv[])
{
  if (argc > 1 &&
      strcmp(argv[1],"-s")==0)
    strip = 1;
  wiz();
  exit(0);
}
```

wiz.c:

```
#include <stdio.h>
#include <ctype.h>
#include "wiz.h"

void wiz() {
  /* wyswietlanie stdin */
  /* z wizualizacja     */
  /* znakow specjalnych */
  int c;

  while ((c = getchar()) != EOF)
    if (isascii(c) &&
        (isprint(c) ||
         c=='\n' ||
         c=='\t' ||
         c==' '))
      putchar(c);
    else if (!strip)
      printf("\\%03o", c);
}
```

# Przykład `wiz`: piąta wersja — operacje na plikach

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

int strip = 0;      /* 1 => usuwanie znakow specjalnych */
void wiz(FILE *fp); /* zmieniony prototyp funkcji */

void wiz(FILE *fp)
{
  int c;

  /* jedyna zmiana w funkcji wiz() polega na
     zamianie wywolania getchar() na getc(fp) */
  while ((c = getc(fp)) != EOF)
    if (isascii(c) &&
        (isprint(c) || c=='\n' || c=='\t' || c==' '))
      putchar(c);
    else if (!strip)
      printf("\\%03o", c);
}

int main(int argc, char *argv[])
{
  int i;
  FILE *fp;
```

```
  while (argc > 1 && argv[1][0] == '-') {
    switch (argv[1][1]) {
    case 's':       /* -s: usuwaj znaki specjalne */
      strip = 1;
      break;
    default:
      fprintf(stderr, "%s: nierozpoznana opcja %s\n", argv[0], argv[1]);
      exit(1);
    }
    argc--;
    argv++;
  }

  if (argc == 1)
    wiz(stdin);
  else
    for (i = 1; i < argc; i++)
      if ((fp=fopen(argv[i], "r")) == NULL) {
        fprintf(stderr, "%s: niedostepny plik %s\n", argv[0], argv[i]);
        exit(1);
      }
      else {
        wiz(fp);
        fclose(fp);
      }

  return 0;
}
```

# Funkcje biblioteki `stdio`

`FILE *fp=fopen(s,mode);`
   otwiera plik i zwraca file pointer, mode: `"r","w","a"`

`int c=getc(fp);`
   zwraca przeczytany znak, `getchar()`⇔`getc(stdin)`

`putc(c,fp);`
   zapisuje znak na pliku, `putchar(c)`⇔`putc(c,stdout)`

`fgets(s,n,fp);`
   czyta do s napis (linię, max n-1 znaków), dodaje `\0`, pomija końcowy `\n`

`fputs(s,fp);`
   zapisuje napis s na pliku, UWAGA: puts dodaje `\n`

`ungetc(c,fp);`
   zwraca znak do ponownego przeczytania (max 1)

`fflush(fp);`
   wyprowadza na wyjście zabuforowane dane

`fclose(fp);`

# Typowe operacje na plikach

```
#include <stdio.h>
main(int argc, char *argv[])
{
  FILE *fp1, *fp2;
  char buf[1024];
  int c;

  fp1 = fopen(argv[1], "r");
  fp2 = fopen(argv[2], "w");

  /* czytanie i pisanie znak po znaku */
  while ((c = getc(fp1)) != '.') /* tez moze byc EOF */
    putc(c, fp2);

  /* czytanie i pisanie calymi wierszami */
  while (fgets(buf, 1024, fp1) != NULL)
    fputs(buf, fp2);

  fclose(fp1);
  fclose(fp2);
}
```

# **Funkcje** `printf, fprintf, sprintf`

```
printf("Wynik: %d pkt na %d, %6.1f%%\n",p,mx,100.0*p/mx);
```
wyświetla wynik punktowy i procentowy z opisem słownym

```
printf("%s %s %d\n", imie, nazwisko, pkt);
```
wyświetla imię, nazwisko, i punkty z domyślnymi szerokościami pola, domyślnie dosunięte w prawo

```
printf("%8.1f %-s",x,(x>9.9?"mmHg":"bar"));
```
drukuje liczbę float z daną szerokością pola i precyzją, oraz nazwę jednostki dosuniętą w lewo

```
printf("%6.*f%%",(x<1.0?2:(x<20.0?1:0)),x);
```
procentowe wyniki wyborów

Funkcje printf, fprint, i sprintf zwracają jako wartość liczbę przesłanych na wyjście bajtów.

# Funkcje `scanf, fscanf, sscanf`

`scanf("a=%d x=%f%n", &a, &x, &n);`
dokonuje konwersji i wczytuje liczby do zmiennych, zwraca liczbę wczytanych elementów, która może być mniejsza niż liczba specyfikacji %, lub EOF; `n` przyjmuje liczbę wczytanych dotąd znaków

`scanf("%13s", buf);`
wczytuje napis znakowy ograniczony spacjami (słowo), max 13 znaków, do tablicy znakowej `buf`, dodaje `\0` na końcu

`scanf("%13c", buf);`
wczytuje ciąg znaków podanej długości do tablicy `buf`, nie dodaje `\0`, traktuje spacje i znaki końca linii jak normalne znaki

`scanf("%2d%2c%*2d%2s%2[0-9]",&i,b1,b2,b3);`
dla ciągu wejściowego `"9876 54 3210"` daje wartości: `i=98`, `b1="76"` (bez `\0`), `b2="32"`, `b3="10"`, (oba zakończone `\0`)

Funkcja `scanf` zwraca liczbę „elementów" pliku wejściowego dopasowanych do podanego formatu, być może 0, lub wartość EOF gdy wystąpił koniec pliku na wejściu przed dopasowaniem czegokolwiek.

# Użycie funkcji `sscanf`

Częstym schematem użycia funkcji `fscanf` jest czytanie danych pełnymi wierszami z pliku, np.:

```
char oper; int x, y;

fscanf(fp, "%d %c %d", &x, &oper, &y);
```

Alternatywnie, i często wygodniej, jest używać funkcji `fgets` do wczytania wiersza z wejścia do bufora, a następnie skanowanie tekstu z bufora funkcją `sscanf`:

```
char buf[1024];
char oper; int x, y;

fgets(buf, 1024, fp);
sscanf(buf, "%d %c %d", &x, &oper, &y);
```

Uwaga: jeśli wiersz danych w pliku zawiera znaki po przeczytanych danych, to w pierwszym przypadku te dane nie zostaną przeczytane. Natomiast jeśli wiersz danych jest dłuższy niż 1024 znaki to nie zostanie do końca przeczytany w przypadku drugim.

Na przykład, następujący fragment kodu oczekuje od użytkownika liczby całkowitej, i sprawdza czy liczba wczytała się poprawnie. Zawiera jednak subtelny błąd, i w przypadku wprowadzenia danych, które nie dadzą się zinterpretować jako liczba, wpadnie w nieskończoną pętlę:

```
char oper; int x, y, odpowiedz;

printf("Podaj wynik dzialania: %d %c %d =\n", x, oper, y);
while (scanf("%d", &odpowiedz) != 1) {
  printf("Podana odpowiedz nie jest liczba calkowita.\n");
  printf("Podaj ponownie wynik dzialania: %d %c %d =\n", x, oper, y);
}
```

Błąd polega na tym, że funkcja `scanf` nie wczytuje błędnych danych, i należy je w jakiś sposób pominąć. Unikamy tego problemu stosując rozdzielenie czytania danych z pliku (`fgets`) i ich dekodowania (`sscanf`):

```
char oper, buf[1024]; int x, y, odpowiedz;

printf("Podaj wynik dzialania: %d %c %d =\n", x, oper, y);
fgets(buf, 1024, stdin);
while (sscanf(buf, "%d", &odpowiedz) != 1) {
  printf("Podana odpowiedz nie jest liczba calkowita.\n");
  printf("Podaj ponownie wynik dzialania: %d %c %d =\n", x, oper, y);
  fgets(buf, 1024, stdin);
}
```

# Przykład: kopiowanie plików

```
#include <stdio.h>
main(int argc, char *argv[])
{
  FILE *fp1, *fp2;
  int c;

  fp1 = fopen(argv[1], "r");
  fp2 = fopen(argv[2], "w");
  while ((c = getc(fp1)) != EOF)
    putc(c,fp2);
}
```

na pozór działa poprawnie, ale co będzie w przypadku jakiegoś błędu, np.:

- niepoprawnej liczby argumentów,
- niepoprawnej nazwy pliku(ów),
- braku pliku źródłowego,
- braku praw dostępu do pliku źródłowego,
- braku prawa do zapisu pliku docelowego,
- niedostępnego dysku jednego z plików,
- braku miejsca na dysku,
- itd.

# Przykład: kopiowanie plików (2)

```
main(int argc, char *argv[]) {                    /* wersja 2: z wykrywaniem bledow */
  FILE *fp1, *fp2;  int c;                        /*            funkcji systemowych */

  if (argc != 3) {
    fprintf(stderr, "%s: wymagane 2 argumenty (podane %d)\n", argv[0], argc-1);
    exit(1);
  }
  if ((fp1 = fopen(argv[1], "r")) == NULL) {
    fprintf(stderr, "%s: blad otwarcia pliku %s do odczytu\n", argv[0], argv[1]);
    exit(2);
  }
  if ((fp2 = fopen(argv[2], "w")) == NULL) {
    fprintf(stderr, "%s: blad otwarcia pliku %s do zapisu\n", argv[0], argv[2]);
    exit(3);
  }
  while ((c = getc(fp1)) != EOF) {
    if (putc(c, fp2) == EOF) {
      fprintf(stderr, "%s: blad zapisu na pliku %s\n", argv[0], argv[2]);
      exit(4);
    }
  }
  if (ferror(fp1) != 0) {
    fprintf(stderr, "%s: blad czytania z pliku %s\n", argv[0], argv[1]);
    exit(5);
  }                                               /* pomijamy zamykanie plikow      */
  exit(0);                                        /*         i bledy z tym zwiazane */
}
```

# Przykład: kopiowanie plików (3)

```
main(int argc, char *argv[]) {                    /* wersja 3: wyswietlane          */
  FILE *fp1, *fp2;  int c;                        /*           komunikaty o bledach */

  if (argc != 3) {
    fprintf(stderr, "%s: wymagane 2 argumenty (podane %d)\n", argv[0], argc-1);
    exit(1);
  }
  if ((fp1 = fopen(argv[1], "r")) == NULL) {
    perror("blad otwarcia pliku do odczytu");
    exit(2);
  }
  if ((fp2 = fopen(argv[2], "w")) == NULL) {
    perror("blad otwarcia pliku do zapisu");
    exit(3);
  }
  while ((c = getc(fp1)) != EOF) {
    if (putc(c, fp2) == EOF) {
      perror("blad zapisu na pliku");
      exit(4);
    }
  }
  if (ferror(fp1) != 0) {
    perror("blad czytania z pliku");
    exit(5);
  }
  exit(0);
}
```

# Przykład: kopiowanie plików (4)

```
#include <errno.h>
int errno;

main(int argc, char *argv[]) {                    /* wersja 4: jawnie formatowane   */
  FILE *fp1, *fp2;  int c;                        /*           komunikaty o bledach */

  if (argc != 3) {
    fprintf(stderr, "%s: wymagane 2 argumenty, podane %d\n", argv[0], argc-1);
    exit(1);
  }
  if ((fp1 = fopen(argv[1], "r")) == NULL) {
    fprintf(stderr,"%s: blad otwarcia do odczytu pliku %s, %s",argv[0],argv[1],strerror(errno));
    exit(2);
  }
  if ((fp2 = fopen(argv[2], "w")) == NULL) {
    fprintf(stderr,"%s: blad otwarcia do zapisu pliku %s, %s",argv[0],argv[2],strerror(errno));
    exit(3);
  }
  while ((c = getc(fp1)) != EOF) {
    if (putc(c, fp2) == EOF) {
      fprintf(stderr, "%s: blad zapisu na pliku %s, %s", argv[0], argv[2], strerror(errno));
      exit(4);
    }
  }
  if (ferror(fp1) != 0) {
    fprintf(stderr, "%s: blad czytania z pliku %s, %s", argv[0], argv[1], strerror(errno));
  ...
```

# Przykład: kopiowanie plików (5)

```
#include <errno.h>
int errno;

#define ERR_EXIT(msg,arg,exitno) \
{ fprintf(stderr, "%s: %s %s, %s\n", prog, msg, arg, strerror(errno));\
  exit(exitno); }

main(int argc, char *argv[]) {                    /* wersja 5: z makrem preprocesora*/
  FILE *fp1, *fp2;  int c;                        /*       do komunikatow o bledach */

  if (argc != 3) {
    fprintf(stderr, "%s: wymagane 2 argumenty (podane %d)\n", argv[0], argc-1);
    exit(1);
  }
  if ((fp1 = fopen(argv[1], "r")) == NULL)
    ERR_EXIT("blad otwarcia do odczytu pliku", argv[1], 2);
  if ((fp2 = fopen(argv[2], "w")) == NULL)
    ERR_EXIT("blad otwarcia do zapisu pliku", argv[2], 3);
  while ((c = getc(fp1)) != EOF) {
    if (putc(c, fp2) == EOF)
      ERR_EXIT("blad zapisu na pliku", argv[2], 4);
  }
  if (ferror(fp1) != 0)
    ERR_EXIT("blad czytania z pliku", argv[1], 5);
  exit(0);
}
```

# Uwagi ogólne o postępowaniu z błędami

- Zmienna `errno` jest ustawiana przez funkcje, które wykrywają i sygnalizują sytuacje nienormalne, lecz nie zmienia wartości gdy wynik działania funkcji jest poprawny.
  - Zatem wartość `errno` może odnosić się do wcześniejszego niż ostatnie wywołania funkcji, albo do późniejszego niż to, o które nam chodzi.
- Jak należy postępować z ewentualnymi błędami w funkcjach:
  `perror`, `strerror`, `fprintf(stderr,...)`?
- Czy jest obowiązkowe testowanie i obsługiwanie absolutnie wszystkich sytuacji nienormalnych?
  Czy próby wybrnięcia z nieoczekiwanych błędów mają w ogóle sens?
- Rozsądna zasada: jeśli wystąpił błąd to jesteśmy w kłopotach; lepiej nie kombinujmy tylko starajmy się wybrnąć w najprostszy możliwy sposób.
  W braku lepszego pomysłu możemy WKZP (wyświetlić komunikat i zakończyć program).
- Inny wniosek: własne funkcje piszmy tak, aby w razie porażki ich użytkownik uzyskał informacje o miejscu i przyczynie powstania błędu i mógł podjąć własne decyzje co do dalszego postępowania.

# Dygresja: preprocesor C

- makrodefinicje (ang. *macro*)

```
#define TAK_NIE(x) (x?"TAK":"NIE")

#define argum (arg1,arg2);
fun argum

#define Celsjusz(Kelvin) Kelvin+273.15              /* porazka */
st_farenheita = Celsjusz(st_kelvina) * 1.8 + 32.0;

#define Celsjusz(Kelvin) (Kelvin+273.15)            /* dobrze  */

#define PI 3.14159265358979323846
```

- pliki włączane „nagłówkowe” (ang. *header files*)

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

#include "modulproc.c" /* formalnie poprawne, ale niestosowane */
```

- kompilacja warunkowa

```
#ifdef COLOR_DISPLAY
display(x_pos,y_pos,Times10,BLUE,WHITE,komunikat);
#else
puts(komunikat);
#endif

#if DEBUG > 5
fprintf(stderr, "DEBUG: d=%d, s=%s\n", d, s);
#endif
```

# Opcje wywołania kompilatora dotyczące preprocesora

-**Dmakro[=wart]** zdefiniuj *makro* preprocesora; jeśli wartość nie występuje to jest 1

-**Umakro** skasuj definicję *makro* jeśli taka istnieje (ale to makro może być ponownie zdefiniowane w programie)

-**P** zatrzymanie kompilacji po preprocesorze, wynik w pliku z końcówką *.i* (kompilator GNU inaczej rozumie tę opcję i wymaga podania opcji -E)

-**E** zatrzymanie kompilacji po preprocesorze, wynik na wyjściu

-**C** pozostawienie komentarzy w programie (uzupełnienie opcji -E)

-**H** powoduje wyświetlanie nazw włączanych plików nagłowkowych

# Użycie tablic

```
#include <stdio.h>
main() {                          /* count digits, white space, others */
  int c, i, nwhite, nother;
  int ndigit[10];

  nwhite = nother = 0;
  for (i = 0; i < 10; ++i)
    ndigit[i] = 0;

  while ((c = getchar()) != EOF)
    if (c >= '0' && c <= '9')
      ++ndigit[c-'0'];
    else if (c == ' ' || c == '\n' || c == '\t')
      ++nwhite;
    else
      ++nother;

  printf("digits = ");
  for (i = 0; i < 10; ++i)
    printf(" %d", ndigit[i]);
  printf(", white space = %d, other = %d\n", nwhite, nother);
}
```

# Tablice znakowe

Język C stosuje konwencję napisów znakowych jako tablic znaków z dodatkowym znakiem ASCII NUL (`'\0'`) na końcu napisu:

```
char s1[16] = "To jest string.";
char s2[]   = "Jak rowniez to.";
```

`s1` i `s2` są oba 16-elementowymi tablicami znakowymi, których 16-tym znakiem jest automatycznie dodawany `'\0'`.

Na zawartościach tablic `s1` i `s2` można wykonywać różne operacje:

```
strcpy(s1, "To inny string.");
for (i=0; i<16; ++i) s2[i] = '.';
```

Tablica `s1` nadal ma na końcu znak NUL, a `s2` nie, ponieważ nie była na niej wykonywana operacja tablicowa (a jedynie operacje na jej poszczególnych pozycjach).

Pisząc programy w C warto konsekwentnie stosować napisy zakończone znakiem NUL, o ile to możliwe, ale zawsze trzeba mieć świadomość obecności tego znaku w tablicy, i np. zostawiać nań miejsce.

# Mały przykład: szkielet budowy preprocesora

```
#include <stdio.h>
#define BUFSIZE 1024

int main(int argc, char *argv[]) {
  FILE *filein, *fileout;

  if (argc > 1) filein = fopen(argv[1], "r");
  else          filein = stdin;
  if (argc > 2) fileout = fopen(argv[2], "w");
  else          fileout = stdout;
  preproc(filein, fileout);
}

void preproc(FILE *in, FILE *out) {
  char buf[BUFSIZE], first[BUFSIZE];

  while (fgets(buf, BUFSIZE, in) != NULL) {
    if (1 == sscanf(buf, "%s", first)) {
      if (first[0] == '#') {                  /* jest dyrektywa preprocesora */
        fprintf(stderr, ">>>> %s", buf);      /* wypuszczamy tylko na stderr */
        continue;
      }
    }
    fputs(buf, out);                          /* pozostale po prostu na out  */
  } /* koniec pliku */
}
```

# Większy przykład: wyszukiwanie napisów

zadanie: program do wyświetlania tych linii z `stdin`, które zawierają określony napis znakowy

schemat:

```
while( jest jeszcze jedna linia danych )
    if( linia zawiera zadany napis znakowy )
        wyświetl ją
```

```
/* getline: get line into s, return length */
int getline(char s[], int lim) {
  int c, i;

  i = 0;
  while (--lim > 0 && (c=getchar()) != EOF && c != '\n')
    s[i++] = c;
  if (c == '\n')
    s[i++] = c;
  s[i] = '\0';
  return i;
}
```

```
/* strindex: return index of t in s, -1 if none */
int strindex(char s[], char t[]) {
  int i, j, k;

  for (i = 0; s[i] != '\0'; i++) {
    for (j=i, k=0; t[k]!='\0' && s[j]==t[k]; j++, k++)
      ;
    if (k > 0 && t[k] == '\0')
      return i;
  }
  return -1;
}
```

Dygresja: porównanie stringów w sensie jednakowej zawartości:

```
if ((strindex(tab1, tab2) == 0) && (strindex(tab2, tab1) == 0))
  printf("tab1 i tab2 maja identyczna zawartosc\n");
else
  printf("tab1 i tab2 roznia sie zawartoscia\n");
```

Kompletujemy rozwiązanie przykładowego problemu:

```
#include <stdio.h>
#define MAXLINE 1000          /* max dlugosc linii wejsciowej */

int getline(char line[], int max);
int strindex(char source[], char searchfor[]);

char pattern[] = "ould";          /* wzorzec do znalezienia */

/* wyszukaj wszystkie linie pasujace do wzorca */
main()
{
  char line[MAXLINE];
  int found = 0;

  while (getline(line, MAXLINE) > 0)
    if (strindex(line, pattern) >= 0) {
      printf("%s", line);
      found++;
    }
  return found;
}
```

# Operacje na tablicach znakowych

Tablice można porównywać w całości, w sensie identyczności:

```
char tab1[] = "ala ma kota",
     tab2[] = "ala ma kota";

printf("tab1 %s tab1\n", (tab1 == tab1) ? "==" : "!=");    /* "==" */
printf("tab1 %s tab2\n", (tab1 == tab2) ? "==" : "!=");    /* "!=" */
```

Jednak nie można tablic w całości podstawiać:

```
tab2 = tab1; /* niedozwolone */
```

Jest to skutek automatycznego potraktowania zmiennej tablicowej jako stałej.

Dowolne operacje można wykonywać na tablicach element po elemencie, np. kopiowanie, porównywanie, oczywiście pod warunkiem zachowania zgodności typów elementów i rozmiarów tablic. Powyższe tablice tab1 i tab2 porównane znak po znaku okażą się takie same.

```
char tab3[] = {'a','l','a',' ','m','a',' ','k','o','t','a'};
```

Tablica `tab3` nie jest taka sama, ma inny rozmiar i zawartość. Dlaczego?

# Tablice jako parametry funkcji

```
int opad_dzienny[365];

fun srednia(int tab[], int liczba);
fun sredniaroczna(int tab[365]);
```

Jednak w odróżnieniu od Pascala, gdy tablica jest argumentem funkcji, przy jej wywołaniu nigdy nie następuje kopiowanie elementów tablicy do procedury mimo, iż w języku C parametry zawsze są przekazywane przez wartość. W rzeczywistości, do procedury przekazywany jest zawsze tylko wskaźnik do tablicy. Dlatego też funkcje mogą jawnie deklarować swój argument jako wskaźnik do elementu. Jest to poprawne, lecz dokładny mechanizm poznamy nieco później.

```
fun srednia(int *tab, int liczba);
fun sredniaroczna(int *tab);
```

# Wskaźniki i podstawowe operacje

operatory referencji & i dereferencji *

```
int i, w, *ip, *jp;

ip = &i;      /* wziecie adresu zmiennej -- tworzy wskaznik;
                 operacja zawsze poprawna dla zmiennych          */
w = *ip;      /* wziecie wartosci zmiennej, do ktorej mamy wskaznik;
                 poprawna o ile wskaznik poprawny                */
```

jeśli ip zawiera wskaźnik do zmiennej x, to zapis *ip może pojawić się wszędzie tam, gdzie może x

```
*ip = *ip + 1; /* zwieksza wartosc elementu *ip */
*ip += 1;      /* tak samo */
++*ip;         /* tak samo */
(*ip)++;       /* tak samo */
```

na wskaźnikach można wykonywać operacje == * &

```
jp = ip;
jp = &ip;      /* jp zawiera wskaznik do zm.wskaznikowej */
w = **jp;      /* teraz bedzie w == *ip, o ile poprawne  */
```

# Wskaźniki jako argumenty funkcji

Podstawowe konstrukcje:

```
int i, *ip;
fun(int x, int *y);
```

```
fun(5, ip);      /* poprawne ! */
fun(i, &i);      /* poprawne ! */
fun(*ip, ip);    /* poprawne ? */
```

Użycie wskaźników w parametrach do przekazywania wartości na zewnątrz:

```
/* f-cja zamienia wartosci argumentow */
void swap(int x, int y) {
  int tmp;

  tmp = x;
  x = y;
  y = tmp;
}

/* wywolanie, niestety, niepoprawne */
swap(a, b);
```

```
/* lepsza wersja */
void swap(int *x, int *y) {
  int tmp;

  tmp = *x;
  *x = *y;
  *y = tmp;
}

/* teraz dziala */
swap(&a, &b);
```

Czy zmienna `tmp` w funkcji `swap` mogłaby też być wskaźnikiem, tzn. czy można w powyższej funkcji zastąpić `tmp` przez `*tmp`?

# Arytmetyka wskaźników

Wskaźniki stanowią dane swojego własnego typu (wskaźnikowego), który jednak jest podobny do typu liczb całkowitych. Wartość wskaźnika można zobaczyć.

```
char ch, *chp;
int  i,  *ip;

chp = 0;   /* inicjalizacja wartością 0 */
ip = &i;   /* inicjalizacja poprawną wartością wskaźnikową */
printf("ip = %d\n", ip);
```

Do wartości wskaźnika można przypisać lub porównać, oprócz normalnych wartości wskaźnikowych, również wartość 0. Można też do wskaźnika dodać (lub odjąć) niewielką liczbę całkowitą, na przykład 1, tworząc wskaźnik do elementu następnego za danym.

```
chp = &c;
chp += 1;  /* chp wskazuje do następnego elementu po c */
ip += 1;   /* ip wskazuje do nastepnego elementu po i */
```

Wartość liczbowa wskaźnika `chp` zwiększyła się o 1, natomiast wskaźnik `ip` zwiększył się o **jakąś** wartość, być może o 4. (W rzeczywistości zwiększył się o liczbę bajtów przypadającą na wartość typu int, różną na różnych systemach.)

# Tablice i wskaźniki

W języku C obowiązuje konwencja, na mocy której można używać wartości zmiennej tablicowej, jako wartości wskaźnika pierwszego elementu tablicy:

```
char s1[20], s2[20];
char *s3, *s4;

s3 = &s1[0];                /* wskaznik do pierwszego elementu */
s3 =  s1;                   /* rownowazne na mocy konwencji  */

if ((s3+1) == &s1[1]) ...; /* z konwencji i arytmetyki wskaznikow */
if (*(s3+1) == s1[1]) ...; /* z powyzszego */
```

W konsekwencji, operacje na tablicach można wykonywać alternatywnie przy użyciu wskaźników, np. kopiowanie:

```
/* kopiowanie tablic */
/* znak po znaku     */

for (i = 0; i < 20; ++i)
  s2[i] = s1[i];
```

```
/* to samo przy uzyciu wskaznikow */
s3 = s1;
s4 = s2;
for (i = 0; i < 20; ++i, ++s3, ++s4)
  *s4 = *s3;
```

# Biblioteka `string`

```
#include <string.h>

char *strcpy(char *dst, const char *src);
char *strncpy(char *dst, const char *src, size_t n);
size_t strlcpy(char *dst, const char *src, size_t dstsize);
char *strdup(const char *s1);
size_t strlen(const char *s);
char *strcat(char *dst, const char *src);
char *strncat(char *dst, const char *src, size_t n);
size_t strlcat(char *dst, const char *src, size_t dstsize);
char *strchr(const char *s, int c);
char *strrchr(const char *s, int c);
int strcmp(const char *s1, const char *s2);
int strncmp(const char *s1, const char *s2, size_t n);
int strcasecmp(const char *s1, const char *s2);
int strncasecmp(const char *s1, const char *s2, int n);
size_t strcspn(const char *s1, const char *s2);
size_t strspn(const char *s1, const char *s2);
char *strpbrk(const char *s1, const char *s2);
char *strtok(char *s1, const char *s2);
char *strstr(const char *s1, const char *s2);
```

# Tablice i wskaźniki — kopiowanie tablic

Pamiętamy, że bezpośrednie przypisywanie sobie jakichkolwiek tablic jest niepoprawne, ponieważ tablice nie są zmiennymi. Kopiowanie tablic zawsze musi się odbywać element po elemencie, np. z użyciem wskaźników:

```
char s1[20], s2[20] = "Ala ma kota.";
char *s3, *s4;

s1 = s2;                    /* niedozwolone */
strcpy(s1, s2);             /* tak mozna, funkcja biblioteczna */
s3 = s1;                    /* tez dozwolone */
strcpy(s4, s3);             /* zle, s4 nie jest tablica */
s4 = s2;                    /* oczywiscie */
strcpy(s4, s3);             /* teraz dobrze, kopiuja sie s2 do s1 */
```

UWAGA: powyższe zależności dotyczą dowolnych tablic. Jednak funkcje biblioteki `string` działają tylko dla tablic znakowych.

# Tablice i wskaźniki — porównywanie tablic

Pamiętamy, że zmienne tablicowe można porównywać operatorami "==" i "!=" dla sprawdzenia identycznej tożsamości (lecz nie zawartości) tablic. Użycie pomocniczych zmiennych wskaźnikowych nie zmienia sensu tych porównań:

```
char s1[20], s2[20] = "Ala ma kota.";
char *s3, *s4;

strcpy(s1, s2);
s3 = s1;
s4 = s2;

if (s1 != s2) printf("s1 != s2\n");
if (s1 == s3) printf("s1 == s3\n");
if (s2 == s4) printf("s2 == s4\n");
if (s3 != s4) printf("s3 != s4\n");

if (strcmp(s1,s2) == 0) printf("strcmp(s1,s2) == 0\n");
if (strcmp(s1,s3) == 0) printf("strcmp(s1,s3) == 0\n");
if (strcmp(s1,s4) == 0) printf("strcmp(s1,s4) == 0\n");
/* i tak dalej, wszystkie maja te sama zawartosc */
```

# Tablice i wskaźniki — przydział pamięci

Pamiętamy, że tablice i wskaźniki mogą być inicjowane stałą wartością:

```
char tab[] = "To jest string.";
char *ptr  = "Jak rowniez to.";
```

Uzyskujemy w ten sposób dwie tablice znakowe, lecz poprzez istotnie różne zmienne. `tab` jest tablicą, której zawartość jest zainicjalizowana określonymi znakami, której nie można zmienić jako zmiennej, ale której wszystkie pozycje znakowe mogą być dowolnie zmieniane. Natomiast `ptr` jest zmienną wskaźnikową zainicjalizowaną wskaźnikiem na napis znakowy. Wartość tej zmiennej wskaźnikowej można zmieniać dowolnie, lecz zawartości pozycji znakowych nie (napis jest tablicą stałą, przydzieloną w pamięci stałych).

```
tab[1] = ptr[1];        /* poprawne kopiowanie znakow */
*(tab+1) = *(ptr+1);    /* rowniez poprawne */
tab = ptr;              /* to przypisanie jest NIEDOZWOLONE */

ptr[1] = tab[1];        /* kopiowanie znakow NIEDOZWOLONE */
*(ptr+1) = *(tab+1);    /* rowniez NIEDOZWOLONE */
ptr = tab;              /* poprawne, choc gubi pamiec */
```

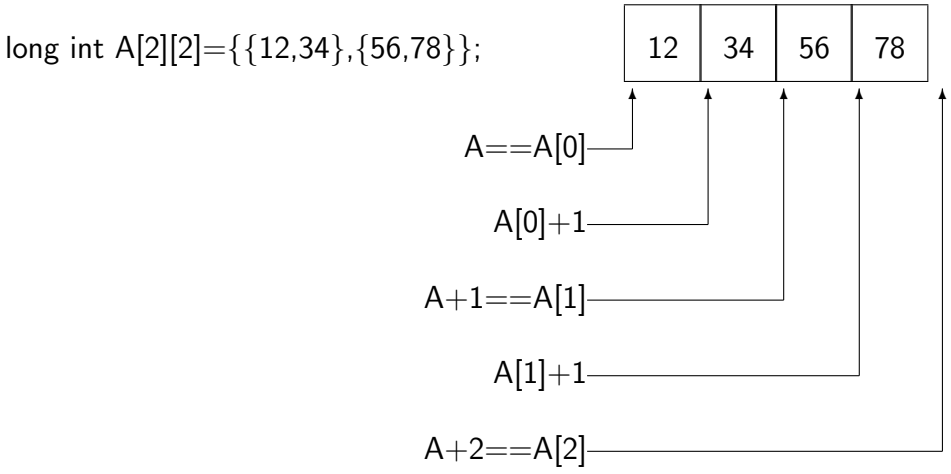
# Przykład: ciąg dalszy budowy preprocesora

```
void preproc(FILE *in, FILE *out) {
  char buf[BUFSIZE], buf2[BUFSIZE], *strptr, *strptr2;
  FILE *in2;

  while (fgets(buf, BUFSIZE, in) != NULL) {
    strptr = buf + strspn(buf, TABSPACE);   /* przeskakujemy "biale" znaki   */
    if (*strptr != '#') {                   /* to jest zwykly wiersz danych  */
      fputs(buf, out);                      /* po prostu wyswietlamy na out  */
      continue;
    }

    /* teraz wiemy, ze mamy wiersz z dyrektywa preprocesora */
    strptr += 1 + strspn(1+strptr,TABSPACE); /* przeskakujemy # i "biale" zn.*/
    if (0 == strncmp(strptr, "include", 7)) { /* mamy "include"-a */
      strptr = strchr(strptr, '"');
      if (strptr != NULL) {
        strptr2 = strchr(strptr+1, '"');
        if (strptr2 != NULL) {
          strncpy(buf2, strptr+1, strptr2-strptr-1);
          buf2[strptr2-strptr-1] = '\0';
          fprintf(stderr, ">>>> Otwieramy plik >>%s<<\n", buf2);
          in2 = fopen(buf2, "r");
          preproc(in2, out);                /* rekurencyjne wywolanie preproc*/
          continue;                         /* wykonane */
        } /* else BLAD */
      } /* else BLAD */
```

```
      fprintf(stderr, ">>> Blad danych, brak nazwy pliku: %s", buf);
    } /* tu jestesmy OK */

    /* aha, czyli jest to jakas nieznana dyrektywa preprocesora */
    fprintf(stderr, ">>>> %s", buf);        /* wypuszczamy tylko na stderr   */

  } /* koniec pliku */
}
```

# Tablice wielowymiarowe

Tablice wielowymiarowe można w języku C tworzyć jako tablice tablic. Taka tablica zajmuje w pamięci jeden blok, gdzie po kolei umieszczone są elementy najniższego poziomu (ostatniego najbardziej na prawo indeksu).




Ze względu na traktowanie nazwy tablicy w wyrażeniu jako wskaźnika do pierwszego elementu pojawiają się pewne charakterystyczne własności tablic wielowymiarowych, np.: `A==A[0]==*A`, a gdyby tablica `A` miała trzy wymiary, to byłoby również: `A==A[0][0]==**A`.

Arytmetyka adresów pozwala tworzyć szereg dalszych konstrukcji.

```
long int A[2][2]={{12,34},{56,78}};

printf("   A=%lu\n", (unsigned long)A);
printf("  *A=%lu     **A=%lu\n", (unsigned long)*A, (unsigned long)**A);
printf("A[0]=%lu   *A[0]=%lu\n",(unsigned long)A[0],(unsigned long)*A[0]);
printf("A[1]=%lu (*A)[1]=%lu\n",(unsigned long)A[1],(unsigned long)(*A)[1]);
printf("A+1 =%lu *(A[1])=%lu\n",(unsigned long)(A+1),(unsigned long)*(A[1]));
printf("(A+1)[1]=%lu\n",(unsigned long)(A+1)[1]);
```

wyniki z komputera 32-bitowego:

```
   A=4290770936
  *A=4290770936     **A=12
A[0]=4290770936   *A[0]=12
A[1]=4290770944 (*A)[1]=34
A+1 =4290770944 *(A[1])=56
(A+1)[1]=4290770952
```

wyniki z komputera 64-bitowego:

```
   A=18446744071562065536
  *A=18446744071562065536     **A=12
A[0]=18446744071562065536   *A[0]=12
A[1]=18446744071562065552 (*A)[1]=34
A+1 =18446744071562065552 *(A[1])=56
(A+1)[1]=18446744071562065568
```

# Struktury

```
struct point {
  int x;
  int y;
}

struct point p1, p2, p3;

struct point p_start = { 2, 3 };

struct rect {
  struct point pt1;
  struct point pt2;
}

struct point makepoint(int x, int y)
{
  struct point temp;

  temp.x = x;
  temp.y = y;
  return temp;
}
```

```
struct rect screen;
struct point middle;

screen.pt1 = makepoint(0, 0);
screen.pt2 = makepoint(XMAX, YMAX);
middle = makepoint((screen.pt1.x + screen.pt2.x)/2,
                   (screen.pt1.y + screen.pt2.y)/2);
```

# Struktury (cd.)

Operacje dozwolone na strukturach:

- branie wartości elementów
- branie adresu struktury (tworzenie wskaźnika)
- przypisanie w całości (de facto kopiowanie)
- wysyłanie jako parametru do procedury (również kopiowanie)
- zwracanie jako wartości funkcji

Niedozwoloną operacją jest porównywanie struktur!

Struktury i tablice mogą być inicjalizowane łącznie listą stałych wartości:

```
struct key {
  char *word;
  int count;
};

struct key keytab[NKEYS] = {
  {"auto", 0},
  {"break", 0},
  {"case", 0},
  {"char", 0},
  /* ... */
  {"while", 0}
}
```

# Alokacja (przydział) pamięci

Zmienne deklarowane na początku bloku, zwane statycznymi, mają automatycznie przydzielaną pamięć (uwaga: nie mylić z klasami alokacji pamięci `static` i `auto`, o których będzie za chwilę) na cały czas istnienia bloku:

```
{ int a,b,c; float x,y,z; ... }
```

Do takich zmiennych odwołujemy się przez ich nazwę, choć w języku C możliwe jest także wzięcie ich wskaźnika (operator &) i odwoływanie się przez wskaźnik. Możliwe jest również tworzenie zmiennych dynamicznych, czyli obszarów pamięci dynamicznej, do których odwoływać się można tylko przez wskaźnik:

```
#include <stdlib.h>
void *malloc(size_t size);
void *calloc(size_t nelem, size_t elsize);
void *realloc(void *ptr, size_t size);
void free(void *ptr);

#include <alloca.h>
void *alloca(size_t size);
```

(Uwaga: funkcja `alloca` nie jest objęta większością standardów i na niektórych systemach nie istnieje, zatem nie powinno się jej używać.)

```
char *charptr, line[500];
fgets(line,500,fp);
charptr = (char *)
        malloc(strlen(line));
strcpy(charptr,line);
```

```
#include <limits.h>
#include <stdio.h>
#define BLOK 1000

void main() {
  char *memptr, *tmpptr;
  size_t memsize = BLOK;

  memptr = (char *) malloc(memsize);
  if (memptr != NULL) {
    do {
      memsize += BLOK;
      tmpptr = (char *)
               realloc(memptr,memsize);
      if (tmpptr != NULL) {
        printf("Mamy %d\n", memsize);
        memptr = tmpptr;
      }
    } while (tmpptr != NULL);
  }
  printf("Nie mamy %d\n", memsize);
  free(memptr);
}
```

# Dynamiczna alokacja pamięci na struktury i tablice

```
#define COUNT 1000
#define STRSIZE 20
struct elem {
  char info[STRSIZE];
  int nr_kolejny;
};

int main() {
  struct elem *struct_ptr, *arr_ptr, *el_ptr;
  size_t el_count = COUNT;
  int i;

  struct_ptr = (struct elem *) malloc(sizeof(struct elem));
  if (struct_ptr != NULL)
    scanf("%d %s", &struct_ptr->nr_kolejny, struct_ptr->info);

}
```

```
#define COUNT 1000
#define STRSIZE 20
struct elem {
  char info[STRSIZE];
  int nr_kolejny;
};

int main() {
  struct elem *struct_ptr, *arr_ptr, *el_ptr;
  size_t el_count = COUNT;
  int i;

  arr_ptr = (struct elem *) calloc(el_count, sizeof(struct elem));
  if (arr_ptr != NULL) {
    el_ptr = arr_ptr;
    for (i=0; i < el_count; i++) {
      sprintf(el_ptr->info, "Element nr %d", i);
      el_ptr->nr_kolejny = i;
      el_ptr++;
    }
  }

  for (i=0, el_ptr=arr_ptr; i<el_count; i++, el_ptr++)
    printf("%d: %s\n",el_ptr->nr_kolejny,el_ptr->info);
}
```

# Dynamiczne struktury danych

```
#define CHUNKSIZE 4096
typedef struct ListElem {
  unsigned char kawalek[CHUNKSIZE];
  struct ListElem *nastepny;
} LISTA;

int push(LISTA **loc);

main() {
  int wyn;
  long int calk = 0;
  LISTA *lst = NULL;

  while (1) {
    if ((wyn = push( & lst )) <= 0) {
      printf("Koniec pracy, alokacja = %d\n", calk);
      exit(0);
    }
    calk += wyn;
    printf("Alokacja %d, kontynuujemy...\n", calk);
  }
```

```
}

int push(LISTA **loc) {
  LISTA *newone;

  newone= (LISTA *) calloc( 1, sizeof(LISTA) );
  if (newone == NULL) return -1;
  newone->nastepny= *loc;
  *loc= newone;
  return sizeof(LISTA);
}
```

# Klasy alokacji pamięci

Obiekty pamięciowe (zmienne) w języku C mogą należeć do jednej z dwóch klas alokacji pamięci: `auto`, albo `static`.

Klasa `auto` jest domyślna w obrębie funkcji, i obiekty tej klasy tworzone są przy wejściu do bloku, w którym są zadeklarowane, oraz automatycznie kasowane w momencie wyjścia z bloku. Specjalnym przypadkiem klasy `auto` jest deklaracja `register`, która deklaruje zmienną jako `auto` i jednocześnie sugeruje by kompilator przydzielił jej jeden z szybkich rejestrów procesora, zamiast zwykłej komórki pamięci. Kompilator może, ale nie musi zastosować się do tej sugestii, jednak do zmiennych `register` nie można stosować operatora referencji & (dereferencję * można stosować jeśli tylko zmienna zawiera poprawny wskaźnik). Zmienne klasy `auto` mogą być inicjalizowane dowolnymi wyrażeniami obliczanymi przy każdym wejściu do bloku (np. wartościami zmiennych, parametrów funkcji).

Klasa `static` obowiązuje zawsze dla zmiennych globalnych (poza funkcjami). Zmienne tej klasy są tworzone raz, i zachowują cały czas swoją wartość, pomimo wychodzenia i ponownego wchodzenia do bloku (albo ponownego wywołania funkcji). Są domyślnie inicjalizowane na 0 i mogą być inicjalizowane wyłącznie wyrażeniami obliczanymi przez kompilator.

Zmienna może mieć tylko jeden specyfikator klasy alokacji pamięci. Takim specyfikatorem jest również technicznie `extern`, który sam nie określa klasy alokacji pamięci, jednak mówi, że alokacja pamięci dla zmiennej jest określona gdzie indziej.

Klasa alokacji pamięci jest innym atrybutem zmiennej niż **zakres**, który określa część programu, w której można odwoływać się do zmiennej. Zmienne zadeklarowane w bloku są zawsze lokalne w tym bloku, a zmienne zadeklarowane poza wszystkimi blokami są globalne w całym module.

# Przykłady: lokalne zmienne `static`

Lokalne zmienne `static` w funkcjach zachowują swoją wartość w kolejnych wywołaniach funkcji, również rekurencyjnych. Są domyślnie inicjalizowane na 0, a jeśli w deklaracji jest inny inicjalizator, to musi być wyrażeniem, które może obliczyć kompilator w czasie kompilacji programu (wyrażenie złożone tylko ze stałych, bez wywołań funkcji).

```
int fun1(...) {
  static int licznik; /* 0 */

  printf("%d-te wywolanie fun\n",
         licznik);
  licznik++;
}
```

```
int robocza(int rozmiar) {
  static char *roboczy=NULL;
  static int rozmiar_roboczy;

  if (roboczy==NULL) {
    roboczy=(char *)malloc(rozmiar);
    rozmiar_roboczy = rozmiar;
  }
  else if (rozmiar>rozmiar_roboczy) {
    roboczy=(char *)realloc(roboczy,
                            rozmiar);
    rozmiar_roboczy = rozmiar;
  }
}
```

# Przykłady: globalne zmienne `static`

W przypadku zmiennych globalnych deklaracja `static` ma inne znaczenie niż dla zmiennych lokalnych. Powoduje ukrywanie takich zmiennych w module, dzięki czemu mamy gwarancję, że zmienna globalna będzie prywatną zmienną danego modułu źródłowego i przy jego linkowaniu z innymi modułami nie wystąpi konflikt przypadkowo zbieżnych nazw.

```
static char bufor[4096];

void wczytaj_do_bufora();

int szukaj_w_buforze();
```

Deklaracji `static` można używać również w odniesieniu do funkcji globalnych i ma ona wtedy takie samo znaczenie jak dla zmiennych globalnych, czyli ukrycie i zarezerwowanie funkcji do użycia tylko w obrębie danego modułu źródłowego.

# Zmienne i funkcje globalne w wielu modułach źródłowych

`fun.h` (plik nagłówkowy modułu funkcji):

```
#include <stdio.h>     /* moze byc potrzebne w deklaracjach */

extern int glob_1;     /* dekl.uzycia zm.globalnej z innego modulu */
int   fun1(int i);                  /* tylko prototypy */
void  fun2(int, int);               /* funkcji eksportowanych */
```

`fun.c` (plik źródłowy modułu funkcji):

```
#include "fun.h"     /* wczytuje tresc pliku naglowkowego */

static int glob_2 = 0;       /* zmienna globalna modulu, nieeksportowana */
int   fun1(int i) { ... }            /* funkcja eksportowana */
void  fun2(int a, int b) { ... }     /* inna funkcja eksportowana */
static float fun3(float x) { ... }   /* funkcja wewnetrzna modulu */
```

prog.c (program główny):

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

int glob_1;          /* rzeczywista deklaracja zm. globalnej glob_1 */
void main() { ... if (fun1(i)) { ... fun2(j,k); exit(0); } ... }
```