Bibliografia

- B. Kernighan, D. Ritchie, *Język C*, WNT 1988
- Niklaus Wirth, *Algorytmy + struktury danych = programy*, WNT 1989

Na początek przykłady:

- Najprostszy program

```
main()
{
}
```

- Drukowanie tekstu

```
main()
{
  printf("Dzień dobry!");
}
```

- Dwie linie na raz

```
main() {
  printf("Pierwsza linia\nDruga linia");
}
```

Typy, operatory, wyrażenia

- identyfikatory

  `x  i  x12  _123  ilosc_iteracji`

- typy i rozmiary danych

typ	rozmiar	min	max
`char`	1	-128	127
`unsigned char`	1	0	255
`int`	2/4	-2147483648	2147483647
`unsigned`	2/4	0	4294967295
`short int`	2	-32768	32767
`long int`	4	-2147483648	2147483647
`float`	4	$1.8 * 10^{-38}$	$3.4 * 10^{38}$
`double`	8	$2.23 * 10^{-308}$	$1.79 * 10^{308}$
`long double`	10	$3.37 * 10^{-4932}$	$1.18 * 10^{4932}$
`char *  int *  void *`	2/4	-	-

Typy, operatory, wyrażenia – c.d.

- stałe

```
0  1  2.5  123.456e-7  0.123E4  123L  0x1F  \027
"tekst"  ""
```

- deklaracje – bezpośrednio po klamerce rozpoczynającej funkcję

```
int i, j, ilosc;
int tablica[10];
char c, linia[1000];
```

- operatory
  - arytmetyczne: `+ - * / %`
  - logiczne
    - relacje: `> < >= <= == !=`
    - łączniki: `&& ||`
  - bitowe: `& | ^ ~ << >>`
  - przypisania: `= += -= *= /= &= |= ^=`
  - zwiększania i zmniejszania: `++ --`
  - inne: `() [] . -> , ?:`

Priorytety operatorów

Operatory	Łączność
`() [] -> .`	lewostronna
`! ~ - ++ -- & * sizeof (typ)`	prawostronna
`* / %`	lewostronna
`+ -`	lewostronna
`<< >>`	lewostronna
`< <= > >=`	lewostronna
`== !=`	lewostronna
`&`	lewostronna
`^`	lewostronna
`|`	lewostronna
`&&`	lewostronna
`||`	lewostronna
`?:`	lewostronna
`= *= /= %= += -= &= ^= |= <<= >>=`	prawostronna
`,`	lewostronna

printf i scanf

Syntaktyka:

```
#include <stdio.h>
int printf(const char *format[, argument, ...]);
int scanf(const char *format[, address, ...]);
```

Specyfikacja formatu:

```
% [flags] [width] [.prec] [F|N|h|l|L] type_char
```

Przykład:

```
printf("%d --- %.2f --- %5.2f", 3, 12.3, 3.4)
```

wyświetli co w cudzysłowie `"3 --- 12.30 ---  3.40"`

Oznaczenia typów:

type_char	typ wartości	format
Numeryczne		
d lub i	Integer	signed decimal integer
o	Integer	unsigned octal integer
u	Integer	unsigned decimal integer
x	Integer	unsigned hexadecimal int (with a, b, c, d, e, f)
X	Integer	unsigned hexadecimal int (with A, B, C, D, E, F)
f	Floating point	signed value of the form [-]dddd.dddd.
e	Floating point	signed value of the form [-]d.dddd or e[+/-]ddd
g	Floating point	signed value in either e or f form
E	Floating point	Same as e; with E for exponent.
G	Floating point	Same as g; with E for exponent if e format used
Znakowe		
c	Character	Jeden znak
s	String pointer	ciąg znaków aż do 0
%	—	znak %
Wskaźniki		
n	Pointer to int	Wpisuje w zmienną liczbę wypisanych znaków
p	Pointer	Wskaźnik XXXX:YYYY lub YYYY

Specyfikacja formatu:

```
% [flags] [width] [.prec] [F|N|h|l|L] type_char
```

Szerokość:

`width`	znaczenie
n	przynajmniej n znaków
0n	przynajmniej n znaków – uzupełnianie zerami
*	argument wskazuje szerokość

Precyzja:

.prec	znaczenie
(brak)	ustawienia domyślne
.0	dla d,i,o,u,x – ustawienia domyślne dla e,E,f bez miejsc dziesiętnych
.n	n znaków lub miejsc dziesiętnych

Specyfikacja formatu:

```
% [flags] [width] [.prec] [F|N|h|l|L] type_char
```

Prefiks:

Prefiks	Format	znaczenie
F	p s	A far pointer
N	n	A near pointer
h	d i o u x X	A short int
l	d i o u x X	A long int
l	e E f g G	A double
L	e E f g G	A long double
L	d i o u x X	An `__int64`
h	c C	A single-byte character
l	c C	A Wide character
h	s S	A single-byte character string
l	s S	A Wide character string

Sekwencje specjalne:

Sequence	Value	Char	What it does
`\a`	0x07	BEL	Audible bell
`\b`	0x08	BS	Backspace
`\f`	0x0C	FF	Formfeed
`\n`	0x0A	LF	Newline (linefeed)
`\r`	0x0D	CR	Carriage return
`\t`	0x09	HT	Tab (horizontal)
`\v`	0x0B	VT	Vertical tab
`\\`	0x5c		Backslash
`\'`	0x27	’	Single quote (apostrophe)
`\"`	0x22	”	Double quote
`\?`	0x3F	?	Question mark
`\O`		any	O=a string of up to three octal digits
`\xH`		any	H=a string of hex digits
`\XH`		any	H=a string of hex digits

printf i scanf – przykład

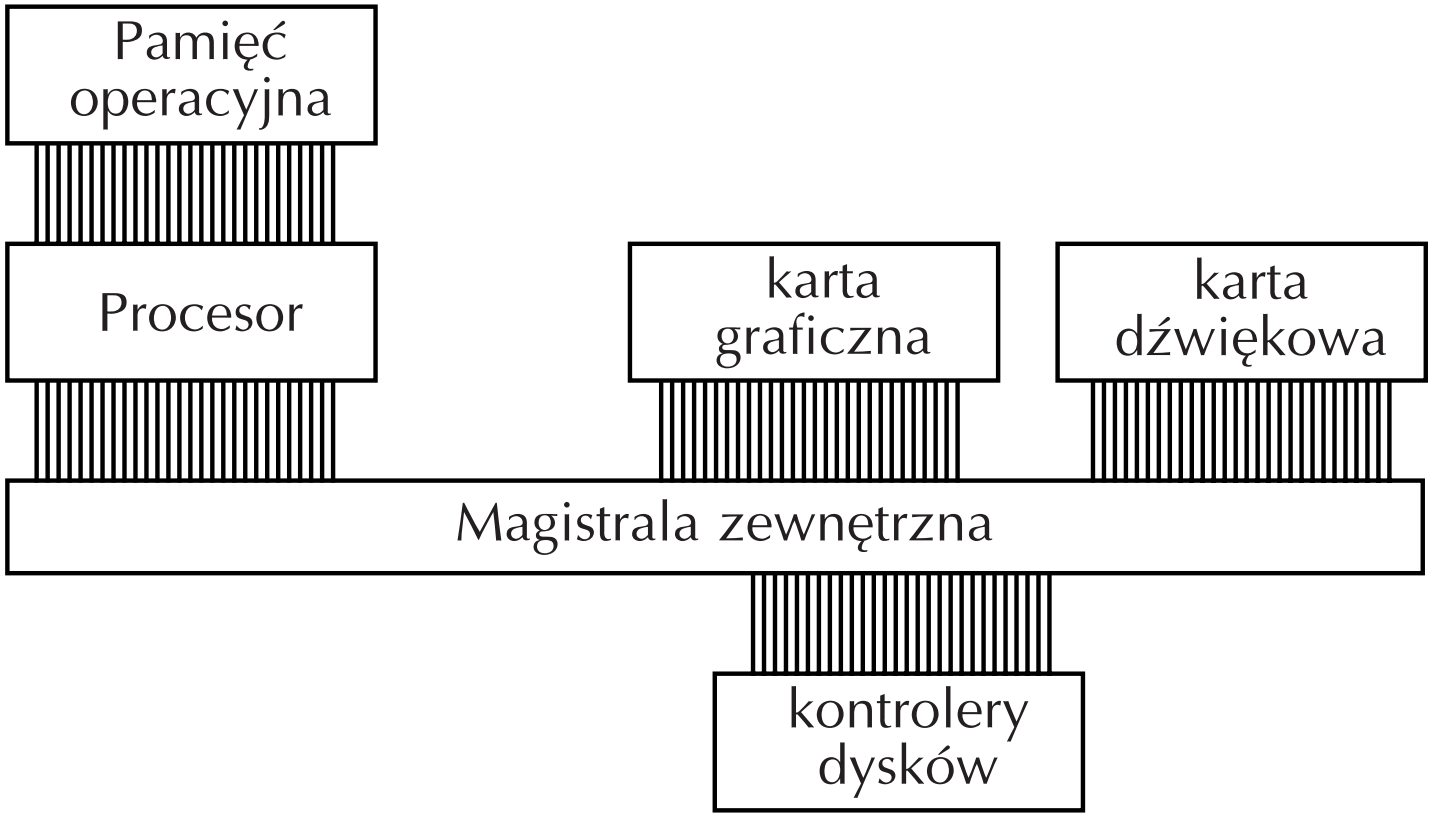
```
#include <stdio.h>

void main()
{
  int n, m=15;
  long k=125L;
  float x, y=12.34;

  printf("%c\n", 65);                          //A
  printf("%d\n", m);                           //15
  printf("%05ld\n", k);                        //00125
  printf("%7.1f\n", y);                        //   12.3

  scanf("%d", &n);                             // wpisujemy 432
  printf("%d\n", n+m);                         //447
  scanf("%f", &x);                             // wpisujemy 1.2
  printf("%f+%6.2f=%.2f\n", x, y, x+y);        //1.200000+ 12.34=13.54
}
```

Pusty kwadrat z gwiazdek: printf, scanf, if oraz for

```
#include <stdio.h>

void main() {
  int n, i, j;

  printf("Podaj rozmiar boku kwadratu: ");
  scanf("%d", &n);

  for (i=0; i<n; i++)
  {
    for (j=0; j<n; j++)
      if (i==0 || j==0 || i==n-1 || j==n-1)
        printf("*");
      else
        printf(" ");
    printf("\n");
  }
}
```

Komputer i jego wykorzystanie

- Sprzęt to ciało – oprogramowanie to dusza
- Komputer – jak pracują krasnoludki



Komputer i jego wykorzystanie – c.d.

- Rodzaje oprogramowania – poziomy od niskiego do wysokiego
  - systemy operacyjne i podstawowe biblioteki obsługi urządzeń
  - oprogramowanie użytkowe i biblioteki użytkowe
  - wysoko wyspecjalizowane biblioteki – programy wysokiego poziomu
- Obsługa zasobów dostępnych w komputerze: pamięć operacyjna, dyski, porty komunikacyjne, wyświetlacze, klawiatury itp.
- Mechanizm przerwań – zgłaszanie przerwań przez urządzenia
- DMA (direct memory access) – bezpośredni dostęp do pamięci

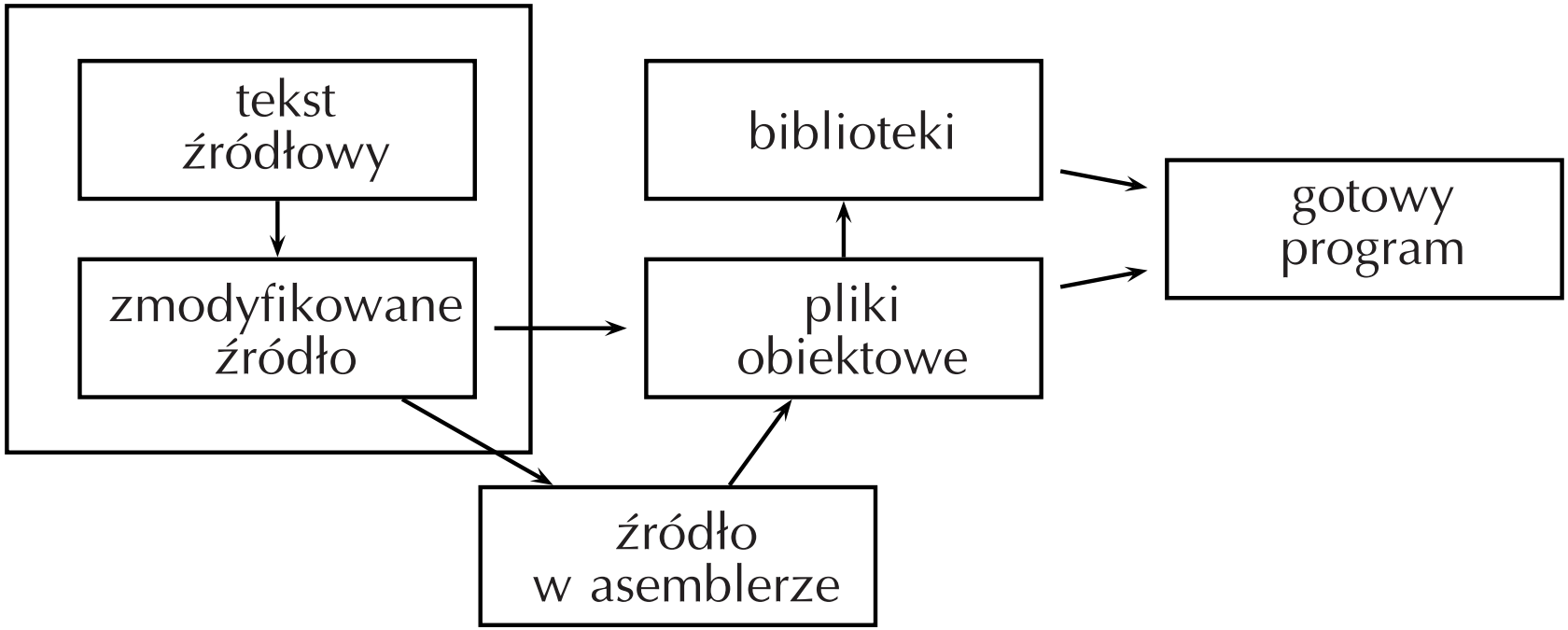
Języki programowania

- Imperatywne
  - liniowe (np. BASIC)
  - strukturalne (np. Pascal, C)
  - obiektowe (np. C++, SmallTalk, Java)
- Deklaratywne
  - logiczne (np. Prolog)
  - funkcyjne (np. ML, Lisp)
- Języki niskiego poziomu (asemblery), wysokiego poziomu (Pascal, C, C++, . . .), języki czwartej generacji (wizualne)
- Jak wybierać język programowania – analiza problemu

Umiejętność programowania

- Znajomość syntaktyki to nie wszystko
- Formatowanie tekstów programów
- Najpierw pomyśl (przynajmniej dwa razy), potem pisz
- Schematy blokowe
- Metody programowania
  - top-down (z góry na dół)
  - bottom-up (z dołu do góry)
- Uwaga na zależności od platformy sprzętowej i systemowej
  - programowanie dla UNIXa
  - programowanie dla DOSa (16 i 32 bitowe, modele pamięci)
  - programowane dla Windows (16 i 32 bitowe)

Cykl życia programu



Biblioteki

- zbiory typów danych i funkcji dla wykonywania pewnej grupy zadań
- korzystający z biblioteki nie musi wiedzieć w jaki sposób ona działa
- proces tworzenia aplikacji staje się prostszy i przez swoją modularność bardziej czytelny i efektywny
- **Uwaga:** Nie przesadzać z uogólnieniami

Pascal i C

- historia Pascala
  - 1971 – Niklaus Wirth – pierwszy raport języka
  - 1974 – N.Wirth & Kathleen Jensen – pierwszy podręcznik
  - Pascal językiem dydaktycznym
  - 1983 – Turbo Pascal firmy Borland Inc.
- historia C
  - 1972 – Dennis Ritchie (Bell Laboratories, New Jersey)
  - rozwój języka – Brian Kernighan & Dennis Ritchie
  - C następcą B (Ken Thompson), B następcą BCPL (1969 – Martin Richards)

Pascal i C

- podobieństwa i różnice na podstawowym poziomie
  - Języki wysokiego poziomu (C również niskiego)
  - Czytelność Pascala i zwięzłość C
  - C daje łatwy dostęp do wszystkich wnętrzności komputera (pamięci, rejestrów itd.)
  - modularność (moduły a biblioteki)
  - C „zrośnięte” z unixem
  - Pascal utożsamia wielkie i małe literki, a C je rozróżnia
  - w C można zrobić wszystko
  - C – „mały język” – istotna rola bibliotek
  - C nie chroni programisty przed nim samym
  - optymalizatory

Pliki nagłówkowe (*.h)

- przeznaczone na definicje stałych, typów i deklaracje funkcji
- nie powinny zawierać kodu (obszary danych i kodu są rozłączne)
- w xyz.h deklaracje typów i funkcji do wykorzystania w innych plikach,
  w xyz.c definicje funkcji, funkcje, typy, stałe i zmienne lokalne dla tego pliku

Systemy liczenia: dwójkowy (binarny), ósemkowy, dziesiętny, szesnastkowy

system	cyfry	przykład	dziesiętnie
dwójkowy	0 1	101	$1 * 2^2 + 1 * 2^0 = 1 * 4 + 1 = 5$
ósemkowy	0 1 2 3 4 5 6 7	270	$2 * 8^2 + 7 * 8^1 = 2 * 64 + 7 * 8 = 184$
dziesiętny	0 1 2 3 4 5 6 7 8 9	309	$3 * 10^2 + 9 * 10^0 = 3 * 100 + 9 = 309$
szesnastkowy	0 1 2 3 4 5 6 7 8 9 A B C D E F	10A	$1 * 16^2 + 10 * 16^0 = 1 * 256 + 10 = 266$

Komputer „myśli” dwójkowo – organizacja pamięci

- 1 **bit** – miejsce na zapamiętanie znaku 0 bądź 1
- 1 **bajt** = 8 bitów – pozwala zapamiętać 2^8 różnych wartości, a więc np. liczby 0-255
- 1 **kB** (kilobajt) = 2^{10} B = 1024 B
- 1 **MB** (megabajt) = 2^{20} B = 1048576 B
- 1 **GB** (gigabajt) = 2^{30} B = 1073741824 B
- 1 **słowo** to ilość informacji przetwarzanej przez procesor w jednym cyklu (procesor 32 bitowy = procesor o słowie 32 bitowym)
- bity i bajty bardziej i mniej znaczące

Binarna reprezentacja danych

- operacje bitowe
  - przesunięcia:

```
00000001 << 3 = 00001000 = 8
00001101 >> 2 = 00000011 = 3
00001101 << 5 = 10100000 = 160
```

  - operacje logiczne: koniunkcja &, alternatywa |, dysjunkcja ^, dopełnienie ~

```
01010101 & 11001100 = 01000100
01010101 | 11001100 = 11011101
00001101 ^ 11001100 = 11000001
~ 01010101 = 10101010
~ 11001100 = 00110011
```

- typowa liczba całkowita zajmuje jedno słowo

Binarna reprezentacja danych – c.d.

- typy zmiennoprzecinkowe (zmiennopozycyjne) – rzeczywiste
  - rozdzielczość i skończoność komputerowych liczb rzeczywistych
  - mantysa i cecha, liczba $=$ mantysa $* 2^{\text{cecha}}$
  - typy 4-bajtowe (3 bajty mantysy i 1 bajt cechy), 6-bajtowe, 8-bajtowe (podwójnej precyzji)
  - należy być ostrożnym przy dodawaniu małej liczby zmiennoprzecinkowej do dużej (mała może zniknąć)
  - wartości, które nie są poprawnymi liczbami (NAN – not a number)
  - standard IEEE 754 – dobry opis pod http://research.microsoft.com/~hollasch/cgindex/coding/ieeefloat.html

Wyrażenie

- stała
- zmienna
- odwołanie do pola struktury
- wywołanie funkcji
- operator z argumentami (argumenty to odpowiednie wyrażenia, operator stosowany odpowiednio do jego specyfiki)
- wyrażenie w nawiasach

Instrukcja

- wyrażenie plus średnik
- instrukcja pusta – sam średnik
- ciąg instrukcji ujęty w klamerki

Instrukcja if-else

```
if (wyrażenie)
  instrukcja-1
else
  instrukcja-2
```

- część `else` jest opcjonalna
- `if (wyrażenie != 0)` najczęściej zastępuje się przez `if (wyrażenie)`
- dwuznaczności

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

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

- dwuznaczność + niepoprawne formatowanie

```
if (n>0)
  for (i=0; i<n; i++)
    if (jest_dobrze(i))
    {
      printf("Hurra! Znalazłem!\n");
      return i;
    }
else
  printf("Błąd - n jest zerem.\n");
```

- Konstrukcje złożone

```
if (wyrażenie)
  instrukcja-1
else if (wyrażenie2)
  instrukcja-2
else if (wyrażenie3)
  instrukcja-3
else
  instrukcja-4
```

Przykład:

```
if (delta>0)
  printf("Są dwa różne pierwiastki...")
else
  if (delta==0)
    printf("Jest jeden pierwiastek podwójny...")
  else
    printf("Nie ma pierwiastków.")
```

Instrukcja switch

```
switch (wyrażenie_całkowite)
{
  case wartość1 : instrukcja1
  case wartość2 : instrukcja2
  .
  .
  .
  case wartośćn : instrukcjan
  default : instrukcja
}
```

- część `default` jest opcjonalna
- wartości przy `case` wskazują miejsce od którego zaczynamy wykonywanie instrukcji – instrukcja `break;` przerywa wykonywanie kolejnych instrukcji

- Przykład:

```
znak = toupper(getchar());
switch (znak)
{
  case ’A’ : Opcja(1); break;
  case ’B’ : Opcja(2); break;
  case ’C’ : Opcja(3); break;
  case ’1’ :
  case ’2’ :
  case ’3’ :
  case ’4’ :
  case ’5’ :
  case ’6’ :
  case ’7’ :
  case ’8’ :
  case ’9’ : Opcja(znak-’0’); break;
}
```

Pętla while

- najpierw sprawdzamy warunek, potem wykonujemy instrukcję

```
while (warunek)
  instrukcja
```

Przykład 1 (suma):

```
int tab[] = {1, 2, 3, 4, 5};
int rozmiar = 5;
int suma = 0, i = 0;
while (i<rozmiar)
  suma += tab[i++];
```

Przykład 2 (konwersacja sieciowa):

```
while (Connected() && ReceiveQuestion())
  SendAnswer();
```

Przykład 3 (czyszczenie bufora klawiatury):

```
#include <conio.h>
...
while (kbhit()) getch();
```

Pętla do-while

- najpierw wykonujemy instrukcję, potem sprawdzamy warunek

```
do
  instrukcja
while (warunek);
```

Przykład 1 (suma):

```
int tab[] = {1, 2, 3, 4, 5};
int rozmiar = 5;
int suma = 0, i=0;
do
  suma += tab[i++];
while (i<rozmiar);
```

Przykład 2 (tak czy nie?):

```
char z;
printf("Tak czy nie? (t/n)");
do
  z = toupper(getch());
while (z != 'T' && z != 'N');
```

Przykład 3 (menu):

```
do
{
  ShowMenu();
  command = GetCommand();
  if (command)
    Run(command);
}
while (command);
```

Pętla for

```
for (wyr1; wyr2; wyr3)
  instrukcja
```

jest równoważna rozwinięciu

```
wyr1;
while (wyr2)
{
  instrukcja
  wyr3;
}
```

Uwaga: Dowolne z trzech wyrażeń można pominąć (brak warunku `wyr2` jest równoznaczny z warunkiem zawsze prawdziwym). W szczególności:

```
for (;;)
{
  ...
}
```

Przykład 1 (suma):

```
int tab[] = {1, 2, 3, 4, 5};
int rozmiar = 5;
int suma, i;

for (i=0,suma=0; i<rozmiar; i++)
  suma += tab[i];
```

Przykład 2 (suma od końca):

```
for (i=rozmiar,suma=0; i--;)
  suma += tab[i];
```

Instrukcja break; – działa także z pętlami

Przykład 1:

```
while (1)
{
  ...
  if (CosNieTak()) break;
}
```

Przykład 2:

```
pierwsza = 1;
for (i=2; i<n/2; i++)
  if (!n%i)
  {
    pierwsza = 0;
    break;
  }
```

Instrukcja continue; – przejście do kolejnej iteracji w pętli

Przykład 1:

```
for (i=0; i<n; i++) {
  if (!przetwarzac(i))
    continue;
  przetwarzaj(i);
}
```

Przykład 2 (rozkład na czynniki pierwsze):

```
    int byljuz = 0;
    printf("%d = ", n);
    for (i=2; n!=1; i++) {
      if (n%i) continue;
      if (byljuz) printf("*");
      else byljuz = 1;
      printf("%d", i);
      n /= i--;
    }
```

Instrukcja goto i etykiety

- **Nie używać!**
- Przykład:

```
for (i=0; i<n; i++)
  for (j=0; j<m; j++)
    if (macierz[i][j] < 0)
      goto znaleziono;

/* co jeśli nie znaleziono */
...

znaleziono:
/* co jeśli znaleziono (na pozycji i, j) */
...
```

Wskaźniki i tablice

- operator & – adres do obiektu (`px = &x`)
- operator * – wartość spod adresu (`x = *px`)
- tablice

```
int a[10];   /* a jest tablicą 10-elementową */
int *pa, x;  /* wskaźnik i zmienna całkowita */
pa = &a[0];  /* równoważne zapisowi pa = a; */
a[1] = 5;
x = *(pa+1); /* równoważne zapisowi x = a[1]; */
```

- `&a[i]` jest równoważne `a+i`
- tablice wielowymiarowe nie istnieją, są tablice tablic:

```
int macierz[5][5];
int *tab1[10];
```

Wskaźniki i tablice – c.d.

- deklaracja tablicy z zainicjowaniem

```
int  a[5] = {1, 2, 3, 4, 5};
char c[] = {'a', 'b', 'c'};
main()
{
  int rozmiar = sizeof(c)/sizeof(*c);
  ...
}
```

Przykład 1 (wczytywanie do tablicy):

```
float tab[10];
int i;
for (i=0; i<10; i++)
{
  printf("tab[%d]: ", i);
  scanf("%f", tab+i);      // można też &tab[i]
}
```

Przykład 2 (sumowanie tablicy przez wskaźnik):

```
float tab[10];
float suma=0;
float *wsk;
...
for (wsk=tab+10; wsk-->tab;)
  suma += *wsk;
```

Struktury i unie

- Syntaktyka struktury (poniżej `nazwa` jest opcjonalna)

```
struct nazwa
{
  lista_deklaracji_pól
};
```

- Przykład deklaracji struktury

```
struct data
{
  int dzien;
  int miesiac;
  int rok;
};
```

- Zmienne typu strukturalnego i wskaźniki do struktur

```
struct data d = {30, 10, 2020}, *pd = &d;
printf("%d-%02d-%d", d.dzien, d.miesiac, d.rok);
printf("%d-%02d-%d", pd->dzien, pd->miesiac, pd->rok);
++pd->dzien              /* zwiększy dzien a nie pd */
```

- Inicjowanie tablicy struktur

```
struct data daty[] = {{1,  1, 2000},
                      {2, 11, 2010},
                      {13, 5, 1612}};
```

- Struktury ze wskaźnikami do samych siebie

```
struct drzewo
{
  float wartosc_wezla;
  struct drzewo *prawe_poddrzewo, *lewe_poddrzewo;
};
```

- Pola bitowe

```
struct
{
  unsigned jest_samcem : 1;
  unsigned jest_ssakiem : 1;
  unsigned ma_rogi : 1;
} flagi;
```

- Unie

```
union opis
{
  int   ival[2];
  float fval;
  void *pval;
} wartosc_opisu;
wartosc_opisu.fval = 1.15;
```

- Deklaracja typedef

```
typedef float odleglosc;
typedef struct
{
  int dzien;
  int miesiac;
  int rok;
} data;

main()
{
  odleglosc d;
  data dzis;
  ...
}
```

Funkcje, argumenty, zmienne

- Syntaktyka definicji funkcji:

```
typ_wartości nazwa(lista_argumentów_jeśli_występują)
deklaracje_argumentów_jeśli_występują
{
  deklaracje_i_instrukcje_jeśli_występują
}
```

lub

```
typ_wartości nazwa(deklaracje_argumentów_jeśli_występują)
{
  deklaracje_i_instrukcje_jeśli_występują
}
```

- Instrukcja `return;` kończy wykonywanie funkcji i ewentualnie zwraca wartość funkcji

- Przykłady:

```
long silnia(n)
int n;
{
  long wynik = 1;
  while (n>0) wynik *= n--;
  return wynik;
}
```

```
long silnia(int n)
{
  long wynik = 1;
  while (n>0) wynik *= n--;
  return wynik;
}
```

- Funkcje rekurencyjne

```
long silnia(int n)
{
  return n<2 ? 1 : n*silnia(n-1);
}
```

- Funkcja `main()` i jej argumenty

```
void main(int argc, char *argv[])
{
  int i;
  for (i=0; i<argc; i++)
    printf("Argument %d = %s\n", i, argv[i]);
}
```

- argumenty przekazywane są zawsze przez wartość

```
void swap(int x, int y)
{
  int z = x;
  x = y;
  y = z;
}
```

```
void swap(int *x, int *y)
{
  int z = *x;
  *x = *y;
  *y = z;
}
```

- argumenty `char napis[]` oraz `char *napis`
- zmienne lokalne i globalne

- lokalne zmienne statyczne

```
void funkcja(void)
{
  static int licznik = 0;
  printf("To jest %d wywołanie tej funkcji\n", ++licznik);
}
```

- wskaźniki do funkcji

```
void mapuj(int tablica[], int dlugosc, int (*funkcja)(int))
{
  int i;
  for (i=0; i<dlugosc; i++)
    tablica[i] = (*funkcja)(tablica[i]);
}
```

Wskaźniki do funkcji i funkcja qsort()

```
#include <stdio.h>     // dla printf()
#include <stdlib.h>    // dla qsort()
#include <string.h>    // dla strcmp() i stricmp()

typedef int (*Funkcja)(char **, char **);

int Porownaj1(char **s1, char **s2) {
  return strcmp(*s1, *s2);
}
int Porownaj2(char **s1, char **s2) {
  return stricmp(*s1, *s2);
}
int Porownaj3(char **s1, char **s2) {
  if (strlen(*s1) > strlen(*s2)) return 1;
  if (strlen(*s1) < strlen(*s2)) return -1;
  return 0;
}
```

```
void main()
{
  int i, j;
  char *ala[] = {"ALA MA KOTA", "Ala ma Kota", "alamakota", "Ala"};
  Funkcja f[] = {Porownaj1, Porownaj2, Porownaj3};
  for (i=0; i<sizeof(f)/sizeof(*f); i++)
    printf("Wynik[%d] = %d\n", i, f[i](ala, ala+1));
  for (i=0; i<sizeof(f)/sizeof(*f); i++)
  {
    qsort(ala, sizeof(ala)/sizeof(*ala), sizeof(*ala), f[i]);
    printf("\nPo sortowaniu nr %d\n", i);
    for (j=0; j<sizeof(ala)/sizeof(*ala); j++)
      printf("%s\n", ala[j]);
  }
}
```

Wejście i wyjście

- Standardowe funkcje realizujące wejście/wyjście:

```
#include <stdio.h>
int putchar(int c);
int getchar(void);
```

```
#include <conio.h>
int putch(int c);
int getch(void);
int getche(void);
```

- Formatowane wejście/wyjście:

```
#include <stdio.h>
int printf(const char *format, ...);
int scanf(const char *format, ...);
```

- Obsługa plików dyskowych – Struktura `FILE`

```
FILE *fp = fopen(nazwaPliku, tryb);
...
fclose(fp);
```

Możliwe tryby otwarcia:

r	otwórz, żeby czytać
w	stwórz, żeby pisać
a	otwórz, żeby pisać (stań na końcu)
r+	otwórz, żeby czytać i pisać
w+	stwórz, żeby czytać i pisać
a+	otwórz, żeby czytać i pisać (stań na końcu)
b	otwórz jako plik binarny
t	otwórz jako plik tekstowy

- Obsługa plików dyskowych – deskryptory plików

```
int handle = open(nazwaPliku, dostep, tryb);
...
close(handle);
```

Flagi dostepu:

`O_RDONLY`	otwórz, żeby czytać
`O_WRONLY`	otwórz, żeby pisać
`O_RDWR`	otwórz, żeby pisać i czytać
`O_APPEND`	po otwarciu stań na końcu pliku
`O_CREAT`	stwórz plik, jeśli nie istnieje
`O_TRUNC`	jeśli plik istnieje, wyczyść zawartość
`O_BINARY`	otwórz jako plik binarny
`O_TEXT`	otwórz jako plik tekstowy

Flagi trybu:

`S_IWRITE`	prawo pisania
`S_IREAD`	prawo czytania
`S_IREAD|S_IWRITE`	prawo pisania i czytania

- Różne funkcje wejścia/wyjścia:

```
int putc(int c, FILE *stream);
int getc(FILE *stream);
int fprintf(FILE *stream, const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
int fputs(const char *s, FILE *stream);
char *fgets(char *s, int n, FILE *stream);

int write(int handle, void *buf, unsigned len);
int read(int handle, void *buf, unsigned len);
```

- Standardowe urządzenia (typu `FILE *`): `stdin, stdout, stderr`

Dyrektywy prekompilatora

- `#include` – wkłada zawartość pliku w miejsce gdzie się pojawia

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

- `#define` – definiuje zmienne prekompilatora

```
#define __PLIK_H
#define TABSIZE 100
#define MAX(x,y) ((x>y)? (x) : (y))
```

- `#ifdef`, `#ifndef` – przetwarzanie warunkowe

```
#ifndef __PLIK_H
#define __PLIK_H
...
#endif
```

Prosta baza danych — plik baza.h

```
#ifndef __BAZA_H
#define __BAZA_H

#define MAX_DL_NAZWISKO 25
#define MAX_DL_IMIE     15

typedef struct {
  char nazwisko[MAX_DL_NAZWISKO+1];
  char imie1[MAX_DL_IMIE+1];
  char imie2[MAX_DL_IMIE+1];
  data dataUr;
} Osoba;

void DodajRekord(int n, Osoba os);
Osoba CzytajRekord(int n);
void ZapiszRekord(int n, Osoba os);
void PokazRekord(int n);
#endif //__BAZA_H
```

Prosta baza danych — plik baza.cpp

```
#include "baza.h"
main()
{
  int wybor, koniec = 0;
  do
  {
    wybor = UruchomMenu();
    switch(wybor);
    {
      case 0 : koniec = 1; break;
      case 1 : DodajRekord(); break;
      ...
    }
  }
  while (!koniec);
}
```

Pamięć przydzielana dynamicznie

Idea: tworzymy wskaźnik (zmienna przechowująca adres) do obiektu, rezerwujemy fragment pamięci i jego adres zapamiętujemy we wskaźniku

- język Pascal

```
var wskaznik = ^Osoba;
begin
  New(wskaznik);     // pamięć przydzielona
  wskaznik^.nazwisko = 'Kowalski';
  ...
  Dispose(wskaznik); // pamięć zwolniona
end.
```

- język C

```
main() {
  Osoba *wskaznik;
  wskaznik = (Osoba *)malloc(sizeof(Osoba));
  strcpy(wskaznik->nazwisko, "Kowalski");
  ...
  free(wskaznik);
}
```

Sterta (ang. heap)

- obszar pamięci przeznaczony na zmienne dynamiczne
- to coś całkiem innego niż stos (ang. stack), który jest rejestrem wywołań funkcji i procedur

Dynamiczna alokacja tablic

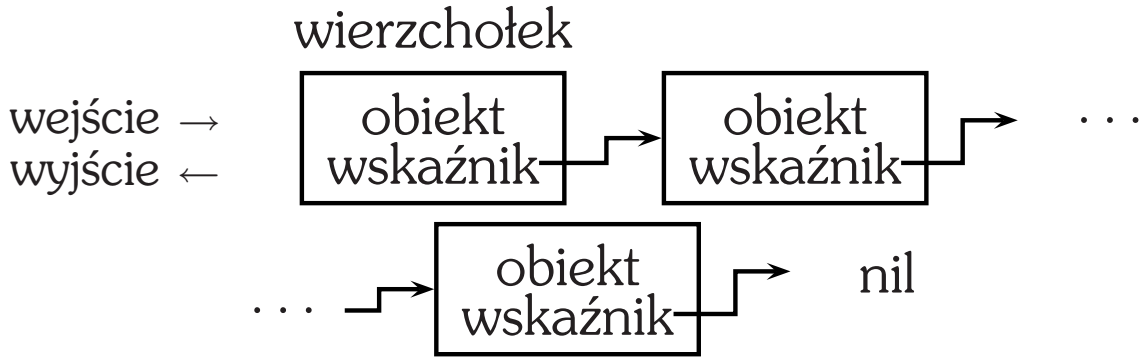
- Obsługa tablic przez wskaźnik do pierwszego elementu
- **Uwaga na rozmiary!** – ważne przy alokacji i przy korzystaniu z tablic
- przykład:

```
void main() {
  int n,i;
  double *wielomian;
  printf("Stopień wielomianu :");
  scanf("%d", &n);
  wielomian = (double *)malloc((n+1)*sizeof(*wielomian));
  for (i=0; i<n+1; i++)
    ...
  free(wielomian);
}
```

- **Uwaga na** `sizeof()`**!**

```
void main() {
  double t1[10];
  double *t2 = (double *)malloc(10*sizeof(double));
  printf("Rozmiar t1 = %d, rozmiar t2 = %d",
    sizeof(t1), sizeof(t2));
  free(t2);
}
```

Dynamiczne struktury danych – stos



- Pascal

```
type wskaznik_stosu = ^skladnik_stosu;
     skladnik_stosu = record
                        obiekt   : typ_obiektu;
                        wskaznik : wskaznik_stosu;
                      end;
```

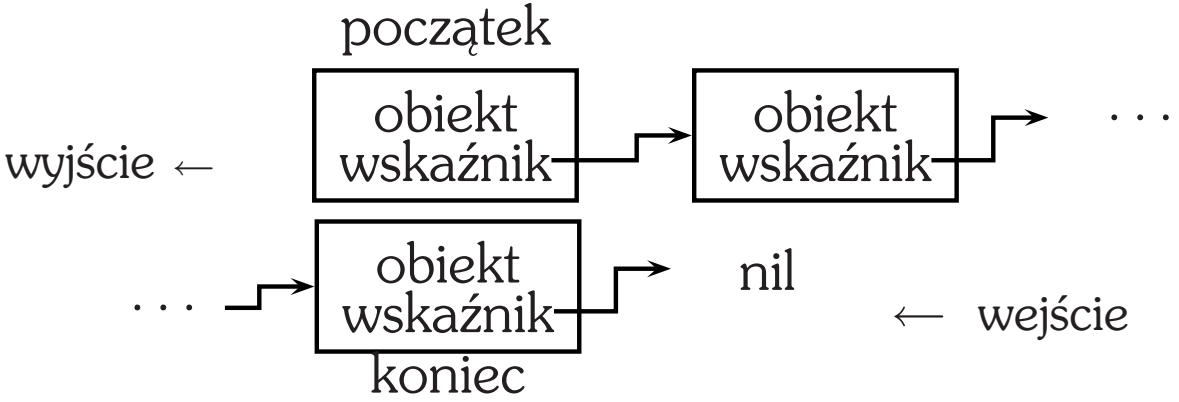
- C

```
typedef struct skladnik_stosu
{
  typ_obiektu obiekt;
  struct skladnik_stosu *wskaznik;
} skladnik_stosu;
```

Obsługa stosu – język C

```
void NaStos(skladnik_stosu **wierzcholek, typ_obiektu o)
{
  skladnik_stosu *tmp = malloc(sizeof(skladnik_stosu));
  tmp->obiekt = o;
  tmp->wskaznik = *wierzcholek;
  *wierzcholek = tmp;
}
typ_obiektu ZeStosu(skladnik_stosu **wierzcholek)
{
  skladnik_stosu *tmp = *wierzcholek;
  typ_obiektu skladnik = tmp->obiekt;
  *wierzcholek = tmp->wskaznik;
  free(tmp);
  return skladnik;
}
```

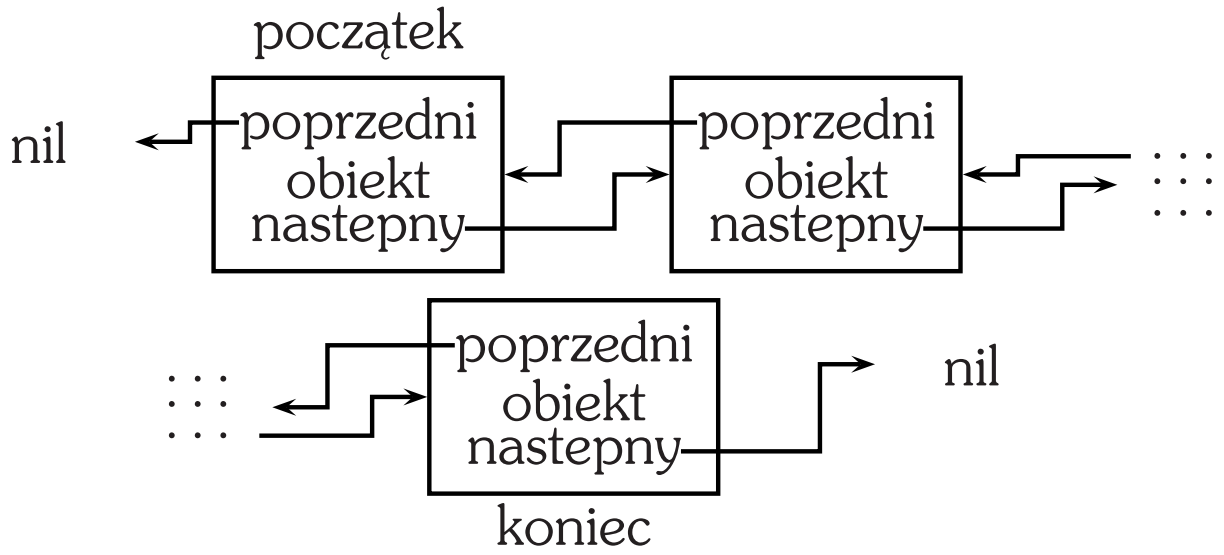
Dynamiczne struktury danych – kolejka




```
typedef struct skladnik_kolejki
{
  typ_obiektu obiekt;
  struct skladnik_kolejki *wskaznik;
} skladnik_kolejki;
```

```
void DoKolejki(skladnik_kolejki **poczatek,
               skladnik_kolejki **koniec, typ_obiektu o) {
  skladnik_kolejki *tmp = malloc(sizeof(skladnik_kolejki));
  tmp->obiekt = o;
  tmp->wskaznik = 0;
  if (*koniec) {
    (*koniec)->wskaznik = tmp;
    *koniec = tmp;
  } else *koniec =  *poczatek = tmp;
}
typ_obiektu ZKolejki(skladnik_kolejki **poczatek,
                     skladnik_kolejki **koniec) {
  skladnik_kolejki *tmp = *poczatek;
  typ_obiektu skladnik = tmp->obiekt;
  if (*koniec == *poczatek) *koniec = 0;
  *poczatek = tmp->wskaznik;
  free(tmp);
  return skladnik;
}
```

Dynamiczne struktury danych – lista dwukierunkowa




```
typedef struct element_listy
{
  typ_obiektu obiekt;
  struct element_listy *poprzedni;
  struct element_listy *nastepny;
} element_listy;
```

```
void DoListyZaElement(element_listy *zaKtory, typ_obiektu o)
{
  element_listy *nowy = malloc(sizeof(element_listy));
  nowy->obiekt = o;
  nowy->poprzedni = zaKtory;
  nowy->nastepny = zaKtory->nastepny;
  nowy->nastepny->poprzedni = nowy;
  zaKtory->nastepny = nowy;
}

void ZListy(element_listy *ktory)
{
  ktory->poprzedni->nastepny = ktory->nastepny;
  ktory->nastepny->poprzedni = ktory->poprzedni;
  free(ktory);
}
```

Inne ważne zagadnienia

- Biblioteki
- Zmienne statyczne (`static`) i zewnętrzne (`extern`)
- Stos i sterta
  - stos (ang. stack) – miejsce na adresy wykonywanych funkcji i ich argumentów
  - sterta (ang. heap) – pamięć do przechowywania danych
- Usuwanie błędów (debugging)
  - narzędzia i środowiska do usuwania błędów
  - makrodefinicja assert
- Przenaszalność oprogramowania
  - uwaga na rozmiary danych
  - uwaga na reprezentację danych (kolejność bajtów)

Lokalne zmienne tablicowe

- zwracanie ich na zewnątrz jest niebezpieczne

```
#include <dir.h>  // dla mktemp()
char *NazwaTymczasowa(void)
{
  char tmp[] = "C:\\TEMP\\TMPXXXXXX";
  mktemp(tmp);
  return tmp;     // Bardzo niebezpieczne
}
```

- rozwiązanie: statyczne tablice, które „żyją” mimo wyjścia z funkcji

```
#include <dir.h>  // dla mktemp()
char *NazwaTymczasowa(void)
{
  static char tmp[] = "C:\\TEMP\\TMPXXXXXX";
  mktemp(tmp);
  return tmp;     // Teraz już w porządku
}
```

Narzędzia programowania

- kompilatory zewnętrzne

```
bcc -c plik1.c plik2.c
gcc -c plik1.c plik2.c
```

- `tlib` – tworzenie i analizowanie bibliotek
- program `make` i pliki `makefile`

```
OBJFILES = error.o rules.o feature.o hidden.o network.o main.o

%o: %cpp
        g++ -c -g $<

rules: ${OBJFILES}
        g++ -o rules ${OBJFILES} -lcurses

clean:
        rm -f rules ${OBJFILES}
```

- profesjonalne edytory (np. `Emacs`, ale i prostsze jak CGenie)
  - automatyczne kopie bezpieczeństwa
  - zarządzanie kompilacją (odnajdywanie błędów itp.)
  - pełna konfigurowalność (rozszerzenia w lispie)
  - współpraca z innymi narzędziami
  - kolorowanie tekstów programu
- `grep` i wyrażenia regularne, `sed` (Unix)

```
grep [Ww]yrazenie *.c *.h
grep program[A-Za-z]* *.txt
grep program[A-Za-z]+ *.txt
```

- `awk` – przetwarzanie plików tekstowych
- `perl`

Narzędzia programowania

- odpluskwianie (usuwanie błędów)
  - srodowiska zintegrowane
  - zewnetrzne debuggery
  - odpluskwianie przy użyciu dwóch komputerów
- `WinSight` – podgląd sygnałów przychodzących do wybranych okien
- Turbo Profiler – szukanie „wąskich gardeł” w programie, liczenie liczby wywołań poszczególnych funkcji, instrukcji itp.
- `lex` i `yacc` (`flex` i `bison`) – definiowanie gramatyk i generowanie kodu źródłowego programu analizy wyrażeń.
- zarządzanie wieloma wersjami programu i praca grupowa
  - RCS – Revision Control System
  - CVS – Concurrent Versions System
  - SCCS – Source Code Control System

Różnice pomiędzy Pascalem a C

- C rozróżnia wielkie i małe litery a Pascal nie
- C jest zarazem językiem niskiego poziomu
- program skompilowany w C jest szybszy
- C jest zwięzły, Pascal jest bardziej czytelny
- w C łatwiej manewrować wskaźnikami, dynamicznie przydzielać pamięć
- w Pascalu dokładniejsza kontrola przed korzystaniem z niedozwolonych obszarów pamięci
- w C są unie
- w C wskaźniki do funkcji, w Pascalu typy proceduralne
- argumenty w C przekazywane zawsze przez wartość, w Pascalu przez wartość lub przez zmienną
- w C jest wygodna i szybka instrukcja `for`
- w C biblioteki, w Turbo Pascalu moduły

Podsumowania i przestrogi

- porządny projekt (przemyślenie problemu)
- nazywanie zmiennych i komentowanie źródeł
- programujemy z góry w dół (top-down)
- funkcje krótkie choć więcej
- kopie bezpieczeństwa
- wieloplatformowość oprogramowania
  - kolejności bajtów w binarnej reprezntacji
  - długości standardowych typów
  - używamy funkcji określonych standardem języka
  - dyrektywa `#pragma`
- nie używamy zmiennych globalnych

Podsumowania i przestrogi – c.d.

- pamięć dynamiczna
  - uwaga na rozmiary przydzielonej pamięci
  - pamiętajmy o jej zwalnianiu
- program budowany z małych kawałków łatwiej przetestować
- dobry programista, to nie ten, który pamięta najwięcej funkcji, ale ten, który wie co daje się zrobić, z łatwością odnajduje funkcje których szuka i potrafi z nich skorzystać.