Programowanie imperatywne

dr inż. Piotr Szwed
Katedra Informatyki Stosowanej
C2, pok. 403

e-mail: pszwed@agh.edu.pl

http://home.agh.edu.pl/~pszwed/

Aktualizacja: 02.05.2020

8. Dynamiczna alokacja pamięci

Dlaczego stosujemy?

Główną motywacją dla stosowania dynamicznej alokacji pamięci jest przetwarzanie danych o rozmiarach, które nie mogą być ustalone w momencie implementacji programu.

- W języku C/C++ brak jest wbudowanych, typów, których rozmiary mogą być ustalane dynamicznie w trakcie wykonania.
- Jeżeli używamy wyłącznie zmiennych globalnych i automatycznych programy mogą przetwarzać dane o ograniczonym rozmiarze i strukturze. Na przykład: macierze o rozmiarach nie większych niż 100x100, pliki tekstowe zawierające nie więcej, niż 1000 wierszy nie dłuższych niż 255 znaków, itd.

```
double a[100][100];
int n,m; // rzeczywiste rozmiary macierzy

struct line {
    int len;
    char text[256];
} textInEditor[1000];
int lineCount;
```

Funkcje (1)

- Dynamiczna alokacja pamięci pozwala na tworzenie obiektów, których rozmiary i liczba nie jest znana w trakcie kompilacji, ale jest ustalana w trakcie wykonania.
- Teoretycznie VLA (tablice o zmiennej wielkości) pozwalają na tworzenie danych o zmiennych rozmiarach, ale rozmiar stosu ogranicza ich wielkość.
- Do obsługi dynamicznie przydzielanej pamięci służą dwie komplementarne funkcje: `malloc()` i `free()` . W C++ ich odpowiednikiem są operatory `new` i `delete`.

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

Przydziela pamięć o rozmiarze size. Zwraca adres przydzielonego obszaru pamięci. Wartość zerowa (NULL) oznacza niepowodzenie.

```
void free( void *memblock );
```

Zwalnia pamięć. Adres będący wartością wskaźnika memblock powinien być rezultatem wcześniejszego wywołania funkcji `malloc`. W zależności od stopnia segmentacji zwolniona pamięć jest pozostawiana do dyspozycji aplikacji lub systemu operacyjnego.

Funkcje (1)

- Dynamiczna alokacja pamięci pozwala na tworzenie obiektów, których rozmiary i liczba nie jest znana w trakcie kompilacji, ale jest ustalana w trakcie wykonania.
- Teoretycznie VLA (tablice o zmiennej wielkości) pozwalają na tworzenie danych o zmiennych rozmiarach, ale rozmiar stosu ogranicza ich wielkość.
- Do obsługi dynamicznie przydzielanej pamięci służą dwie komplementarne funkcje: `malloc()` i `free()` . W C++ ich odpowiednikiem są operatory `new` i `delete`.

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

Przydziela pamięć o rozmiarze size. Zwraca adres przydzielonego obszaru pamięci. Wartość zerowa (NULL) oznacza niepowodzenie.

```
void free( void *memblock );
```

Zwalnia pamięć. Adres będący wartością wskaźnika memblock powinien być rezultatem wcześniejszego wywołania funkcji `malloc`. W zależności od stopnia segmentacji zwolniona pamięć jest pozostawiana do dyspozycji aplikacji lub systemu operacyjnego.

Funkcje (3)

- Dodatkowe funkcje służące do alokacji pamięci (rzadko używane):

```
void * calloc ( size_t num, size_t size );
```

alokuje pamięć dla `num` elementów o rozmiarze `size` i wypełnia blok pamięci zerami.

```
void * realloc ( void * ptr, size_t size );
```

Zmienia rozmiary wcześniej przydzielonego dynamicznie bloku pamięci wskazywanego przez `ptr`. Funkcja może być użyta także do zwalniania pamięci (jeżeli `size` wynosi 0). Zazwyczaj pamięć jest rozszerzana. Funkcja może przenosić blok kopiując zawartość lub po prostu go rozszerzać (jeżeli był ostatni)

Sterta(1)

Pamięć przydzielana jest na tzw. stercie (ang. heap).

- Rozmiar sterty dostępnej dla aplikacji może być bardzo duży. W większości systemów operacyjnych może przewyższać pamięć fizyczną maszyny.
- Rzeczywiste zużycie pamięci sterty może być znacznie większe niż wielkość pamięci przydzielanej dla obiektów aplikacji. Zarządzanie stertą zawsze jest związane z dodatkowym narzutem.
- Funkcje `malloc` i `free` zarządzają listą bloków. Każdy z bloków zawiera nagłówek definiujący wielkość bloku, status (wykorzystywany lub zwolniony), dodatkowe bajty wyrównujące, wskaźniki na następny i poprzedni blok oraz informacje dodatkowe wykorzystywane w trybie uruchamiania.
- Alokacja pamięci dla pojedynczych zmiennych na ogół jest bardzo nieefektywna.

Sterta(2)

Przykład

```
#include <stdlib.h>
int main()
{
	int*i1=(int*)malloc(sizeof(int));
	int*i2=(int*)malloc(sizeof(int));
	printf("i1=%d, i2=%d\n",i1,i2);
	free(i1);
	free(i2);
	return 0
}
```

```
i1=7933632, i2=7933584 (tryb Debug 48B/4B)
i1=7737056, i2=7737040 (tryb Release 16B/4B)
```

Przydział pamięci dla tablic

Przydział pamięci dla n-elementowej tablicy elementów TYPE

```
TYPE*ptr;
ptr = (TYPE*)malloc(n*sizeof(TYPE)) ;
…
free(ptr);
```

W języku C rzutowanie na (TYPE*) można pominąć.

W C++ składnia jest prostsza, kompilator oblicza `n*sizeof(TYPE)` oraz dokonuje rzutowania.

```
TYPE*ptr;
ptr = new TYPE[n];
…
delete []ptr;
```

Uwaga

- Nie należy mieszać wywołań `new→free` oraz `malloc→delete` nawet w przypadku prostych typów. Operatory `new` i `delete` wołają na ogół funkcje `malloc` i `free`, ale nie jest to zagwarantowane.
- Funkcje standardowych bibliotek C zawsze wywołują `malloc`. Alokowanej pamięci nie wolno zwalniać za pomocą `delete`.

```
char * ptr;
ptr = strdup("Ala ma kota"); // tworzy duplikat
...
free(ptr); // nie delete []ptr; !!!
...
char* mystrdup(const char*txt)
{
        char*ptr;
        if(!txt)return 0;
        ptr=(char*)malloc( (strlen(txt)+1)*sizeof(char));
        if(!ptr)return 0;
        strcpy(ptr,txt);
        return ptr;
}
```

Przykład – rozszerzanie tablicy (malloc)

```
int main() {
       char*a="Ala ma ";
       char*b="kota";
       char*text=0,*tmp=0;

       text = (char*)malloc(strlen(a)+1);
       if(!text){ /* błąd */}
       strcpy(text,a);
       printf("%s\n",text);

       tmp=(char*)malloc(strlen(text)+strlen(b)+1);
       strcpy(tmp,text);
       strcat(tmp,b);

       free(text);
       text=tmp;
       printf("%s\n",text);
       free(text);
       return 0;
 }
```

Przykład – rozszerzanie tablicy (realloc)

```
int main() {
    char*a="Ala ma ";
    char*b="kota";
    char*text=0,*tmp=0;

    text = (char*)malloc(strlen(a)+1);
    if(!text){ /* błąd */}
    strcpy(text,a);
    printf("%s\n",text);

    text=(char*)realloc(text,strlen(text)+strlen(b)+1);
    strcat(text,b);

    printf("%s\n",text);
    free(text);
    return 0;
}
```

Przykład – tablice (1)

```
double*allocTable(int size)
{
    double *r = (double*)malloc(sizeof(double)*size);
    return r;
}
double*duplicateTable(double*d,int size)
{
    int i;
    double *r = (double*)malloc(sizeof(double)*size);
    for(i=0;i<size;i++)r[i]=d[i];
    return r;
}
double*extendTable(double*d,int oldsize,int newsize)
{
    int i;
    double *r = (double*)malloc(sizeof(double)*newsize);
    for(i=0;i<oldsize && i<newsize;i++)r[i]=d[i];
    for(;i<newsize;i++)r[i]=0;
    free(d);
    return r;
}
```

Przykład – tablice (2)

```
void printTable(double*d,int size)
{
    int i;
    for(i=0;i<size;i++)printf("%f ",d[i]);
    printf("\n");
}

int main()
{
    int i;
    double*d1,*d2;
    d1=allocTable(10);
    for(i=0;i<10;i++)d1[i]=i;
    d2=duplicateTable(d1,10);
    printTable(d2,10);
    free(d1);
    d2=extendTable(d2,10,30);
    for(i=10;i<30;i++)d2[i]=i;
    printTable(d2,11);
    free(d2);
    return 0;
}
```

Inna wersja extendTable

```
double*extendTable(double*d,int oldsize,int newsize)
{
    int i;
    double *r = (double*)malloc(sizeof(double)*newsize);
    for(i=0;i<oldsize && i<newsize;i++)r[i]=d[i];
    for(;i<newsize;i++)r[i]=0;
    free(d);
    return r;
}
```

```
double*extendTable(double*d,int oldsize,int newsize)
{
    int i;
    d = (double*)realloc(d,sizeof(double)*newsize);
    for(;i<newsize;i++)d[i]=0;
    return d;
}
```

Znika tworzenie nowej tablicy, kopiowanie (jeżeli jest to konieczne - realloc skopiuje bloki pamięci).
Ale w języku C++ ze względu na operator new będzie użyta pierwsza wersja.

Przykład Macierz dwuwymiarowa VLA

Jeżeli nasz kompilator implementuje VLA możemy przetwarzać tablice dwuwymiarowe bez dodatkowych struktur danych.

Wskaźnik `double (*m)[cols]` to wskaźnik do tablicy (wiersza) liczącej `cols` elementów. Czyli `m[0]` adresuje pierwszy wiersz, `m[1]` kolejny, itd.

```
void*create_matrix(int rows,int cols){
    double (*m)[cols]=malloc(rows*cols*sizeof(double));
    for(int i=0;i<rows;i++){
        for(int j=0;j<cols;j++){
            m[i][j]=rand()%100/10.0;
        }
    }
    return m;
}

void print_matrix(int rows,int cols,double(*m)[cols]){
    for(int i=0;i<rows;i++){
        for(int j=0;j<cols;j++){
            printf("%f ",m[i][j]);
        }
        printf("\n");
    }
}
```

Przykład Macierz dwuwymiarowa VLA

```
int main(){
    int rows = 5;
    int cols = 5;
    double (*m)[cols] = create_matrix(rows,cols);
    m[2][3]=-5;
    printf("%p %p %f\n",&m[2][3], (double*)m+2*cols+3,m[2][3]);
    print_matrix(rows,cols,m);
    free(m);

}
```

```
0x8000389c8 0x8000389c8 -5.000000
3.300000 4.300000 6.200000 2.900000 0.000000
0.800000 5.200000 5.600000 5.600000 1.900000
1.100000 5.100000 4.300000 -5.000000 0.800000
9.300000 3.000000 6.600000 6.900000 3.200000
1.700000 4.700000 7.200000 6.800000 8.000000
```

Czyli, aby obliczyć adres elementu w wybranym wierszu i kolumnie, np. `&m[2][3]` kompilator w rzeczywistości oblicza `(double*)m+2*cols+3` w ciągłym obszarze pamięci – stąd rzutowanie na `(double*)`.

Przykład Macierz dwuwymiarowa VLA

Niestety w VisualStudio…

Microsoft nie planuje implementacji VLA, która jest niezgodna ze standardem C++.

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   void* create_matrix(int rows, int cols) {
5       double(*m)[cols] = malloc(rows * cols * sizeof(double));
6       for (int i =
7           for (int
8               m[i]                                   0;
9           }
10      }
11      return m;
12  }
13
14  void print_matrix(int rows, int cols, double(*m)[cols]) {
15      for (int i = 0;i < rows;i++) {
16          for (int j = 0;j < cols;j++) {
17              printf("%f ", m[i][j]);
```

int cols

Wyszukaj w trybie online

wyrażenie musi mieć stałą wartość

Wyszukaj w trybie online

100 % | 3 | 0

Lista błędów

Całe rozwiązanie | 11 Błędy | 0 Ostrzeżenia | 0 Komunikaty | Kompilacja + IntelliSense | Przeszukaj listę błędów

Kod	Opis	Projekt	Plik	W...
E0028	wyrażenie musi mieć stałą wartość	Matrix	Matrix.c	5
E0411	parametr jest niedozwolony	Matrix	Matrix.c	14
E0028	wyrażenie musi mieć stałą wartość	Matrix	Matrix.c	26
C2057	oczekiwano stałego wyrażenia	Matrix	Matrix.c	5

Przeliczanie indeksów

```
void*create_matrix(int rows,int cols){
    double *m = malloc(rows*cols*sizeof(double));
    for(int i=0;i<rows*cols;i++){
            m[i]=rand()%100/10.0;
    }
    return m;
}
void print_matrix(int rows,int cols,double*m){
    for(int i=0;i<rows;i++){
        for(int j=0;j<cols;j++){
            printf("%f ",m[i*cols+j]);
        }
        printf("\n");
    }
}
int main(){
    int rows = 5;
    int cols = 5;
    double *m = create_matrix(rows,cols);
    print_matrix(rows,cols,m);
    free(m);
}
```

Wartości losowe wpisujemy do tablicy jednowymiarowej

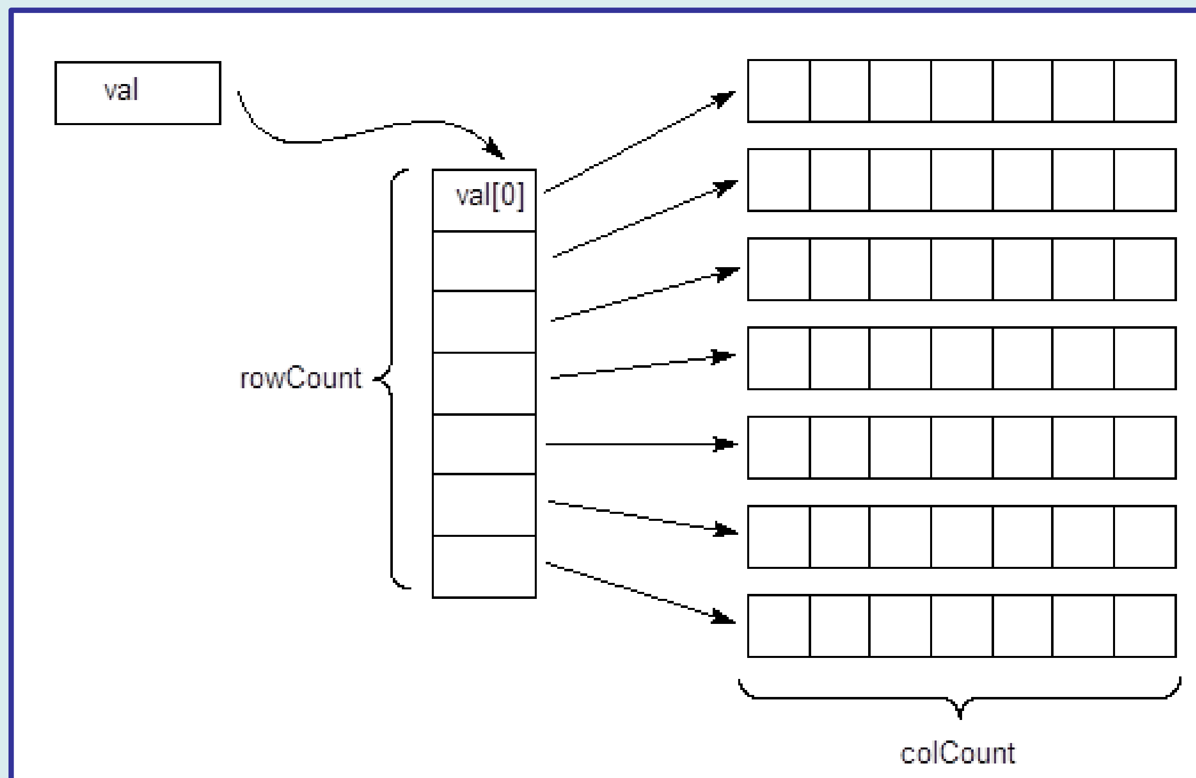
Zamiast `m[i][j]` adresujemy element jako: `m[i*cols+j]`

Raczej należy przyjąć taką strategię programując dla GPU (kart graficznych) w CUDA lub OpenCL.

Przykład: Macierz (1)

```
typedef struct tagMatrix
{
        int rowCount;
        int colCount;
        double**val;
}Matrix;
```

Tablica wskaźników do poszczególnych wierszy macierzy.
Potencjalnie wiersze mogą być różnej długości.



Przykład: Macierz (2)

```
int allocStorage(Matrix*matrix)
{
        int i,j;
        if(matrix->rowCount<=0)return 0;
        if(matrix->colCount<=0)return 0;
        matrix->val = (double**)malloc(
                matrix->rowCount*sizeof(double*));
        if(!matrix->val)return 0;
        for(i=0;i<matrix->rowCount;i++){
                matrix->val[i]=(double*)malloc(
                matrix->colCount*sizeof(double));
        }

        for(i=0;i<matrix->rowCount;i++)
                for(j=0;j<matrix->colCount;j++)
                        matrix->val[i][j]=0;
        return 1;
}
```

Zakłada się, że pola `rowCount` i `colCount` zawierają rozmiary tablicy

Wpierw przydzielana jest pamięć dla tablicy wskaźników do wierszy, następnie dla wierszy…

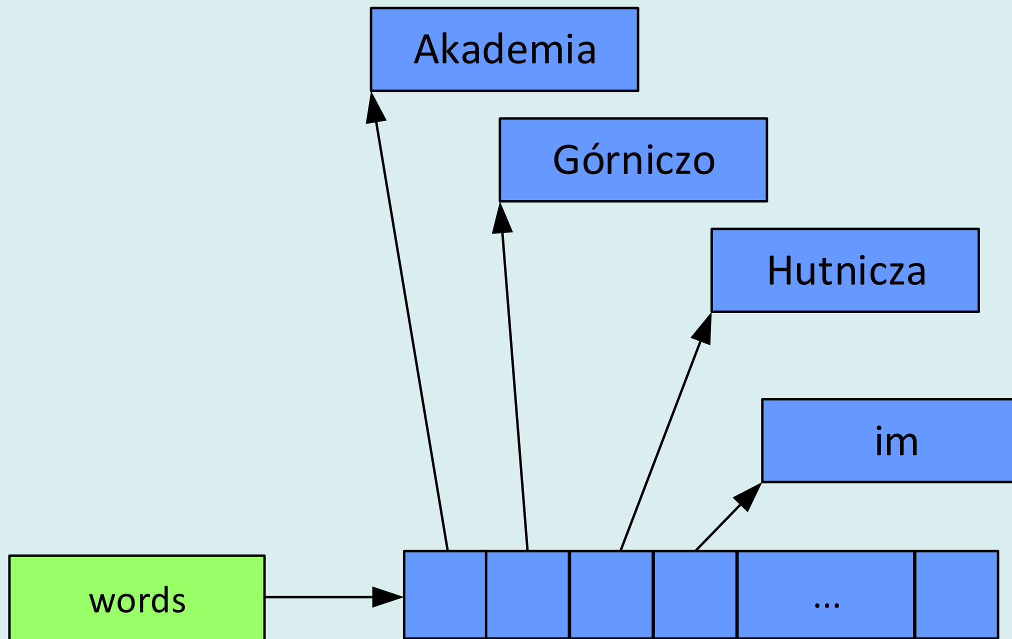
Przykład: Macierz (3)

```
void freeStorage(Matrix*matrix)
{
        int i;
        if(!matrix->val)return;
        for(i=0;i<matrix->rowCount;i++){
                if(matrix->val[i])free(matrix->val[i]);
        }
        free(matrix->val);
        matrix->val=0;
}

int main()
{
        Matrix m;
        m.rowCount = 200;
        m.colCount = 12;
        allocStorage(&m);
        // dostęp do elementów  m.val[row][col]
        //…
        freeStorage(&m);
        return 0;
}
```

Wpierw zwalniane są wiersze, następnie tablica wskaźników do wierszy…

Przykład – rozszerzanie tablicy wskaźników

- Zadaniem jest podzielenie tekstu na słowa.
- Docelowo mamy otrzymać tablicę wskaźników typu `char*`. Każdy element tablicy ma wskazywać wydzielone słowo.
- Pamięć dla słów ma być przydzielona na stercie. Oczywiście, słowa mogą mieć różną długość…



Przykład – rozszerzanie tablicy wskaźników

```
char**tokenize_add_sentinel(char*txt, const char*delim){
    char**words=0;
    int cnt=0;
    for(char*ptr=strtok(txt,delim);ptr;ptr=strtok(NULL,delim)){
        // przedłuż tablicę o 1
        words = (char**)realloc(words,(cnt+1)*sizeof(char*));
        // dodaj kopię tekstu wskazanego przez ptr
        words[cnt]=strdup(ptr);
        cnt++;
    }
    // dodaj na końcu „wartownika” - wskaźnik zerowy
    words = (char**)realloc(words,(cnt+1)*sizeof(char*));
    words[cnt]=0;
    return words;

}
```

Przykład – rozszerzanie tablicy wskaźników

```
int main(){
    char text[]="Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie, "
                "AGH (dawniej Akademia Górnicza w Krakowie), nazwa międzynarodowa "
                "AGH University of Science and Technology (dawniej University of "
                "Mining and Metallurgy) – największa polska uczelnia techniczna, "
                "powołana 8 kwietnia 1919 uchwałą Rady Ministrów. Jedna "
                "z najlepszych uczelni w Polsce.";
    char**words = tokenize_add_sentinel(text,".,()-- ");
    // wydrukuj
    for(char**ptr=words;*ptr;ptr++){
        printf("%s\n",*ptr);
    }
    // zwolnij pamięć
    for(char**ptr=words;*ptr;ptr++){
        free(*ptr);
    }
    free(words);
}
```

```
Akademia
Górniczo
Hutnicza
im
Stanisława
Staszica
w
Krakowie
AGH
dawniej
Akademia
Górnicza
w
Krakowie
nazwa
międzynarodowa
AGH
University
of
Science
and
Technology
dawniej
University
of
Mining
…
```

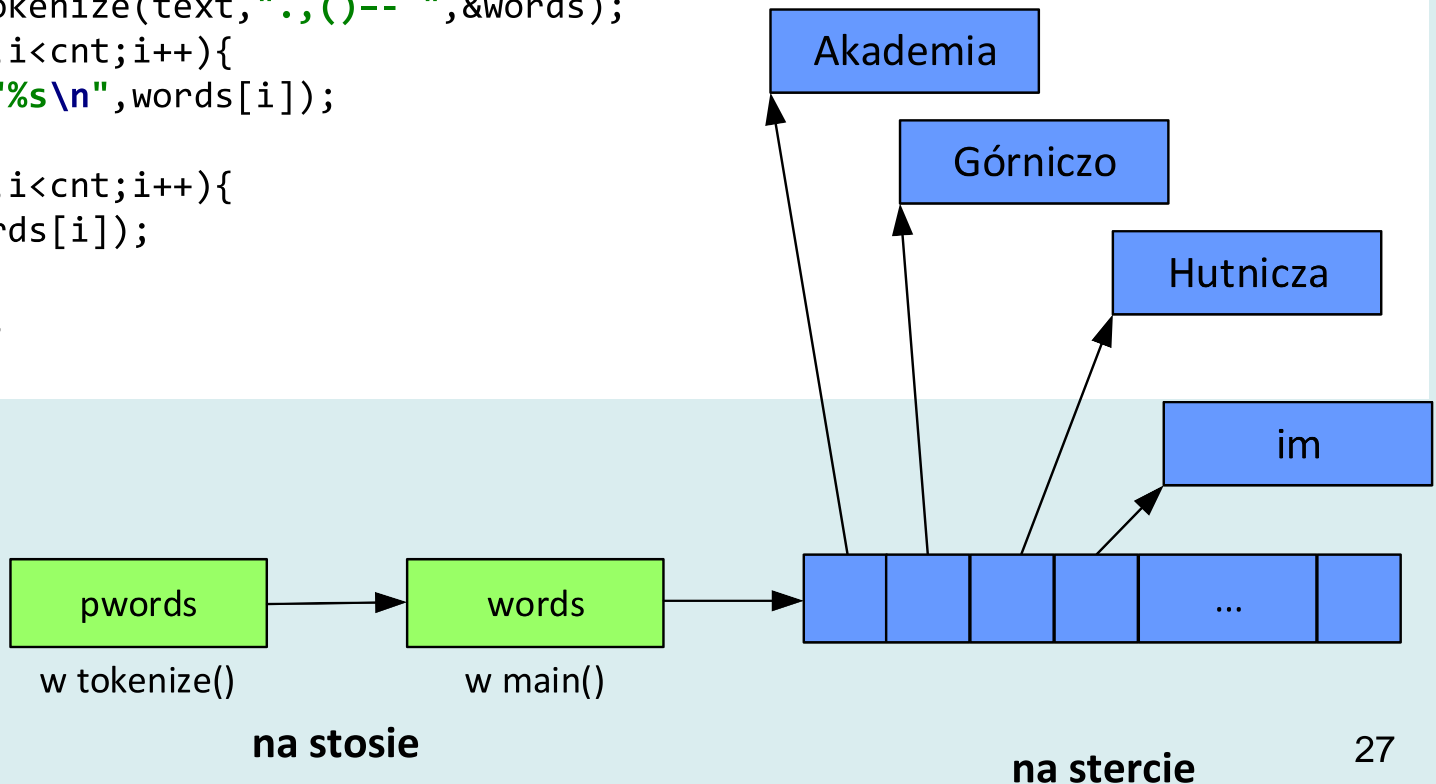
Przykład – inna wersja

```
int tokenize(char*txt, const char*delim,char***pwords){
    *pwords=0;
    int cnt=0;
    for(char*ptr=strtok(txt,delim);ptr;ptr=strtok(NULL,delim)){
        // przedłuż tablicę o 1
        *pwords = (char**)realloc(*pwords,(cnt+1)*sizeof(char*));
        // dodaj kopię tekstu wskazanego przez ptr
        (*pwords)[cnt]=strdup(ptr);
        //nie *words[cnt]=strdup(ptr) -> priorytety operatorów
        cnt++;
    }
    return cnt;
}
```

Funkcja zwraca liczbę wydzielonych elementów (`cnt`). Nie oznacza więc końca tablicy wskaźników zerem.
Parametrem funkcji jest `char***`, czyli wskaźnik do wskaźnika do wskaźnika.
W tym przypadku intencją jest wskaźnik do wskaźnika, który wskazuje pierwszy element tablicy wskaźników.

Przykład – inna wersja

```
int main(){
    char text[]="Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie, "
                "AGH (dawniej Akademia Górnicza w Krakowie), nazwa międzynarodowa "
                "AGH University of Science and Technology (dawniej University of "
                "Mining and Metallurgy) – największa polska uczelnia techniczna, "
                "powołana 8 kwietnia 1919 uchwałą Rady Ministrów. Jedna "
                "z najlepszych uczelni w Polsce.";
    char**words;
    int cnt = tokenize(text,".,()-- ",&words);
    for(int i=0;i<cnt;i++){
        printf("%s\n",words[i]);
    }
    for(int i=0;i<cnt;i++){
        free(words[i]);
    }
    free(words);
}
```



Optymalizacja

Często alokacja pamięci dla pojedynczych elementów jest nieopłacalna – zwłaszcza jeżeli alokacje dla różnych wskaźników przeplatają się.

```
int one_alloc(){
    int*w1=0;
    int*w2=0;
    clock_t s = clock();
    for(int i=0;i<10000000;i++){
        w1=realloc(w1,(i+1)*sizeof(int));
        w2=realloc(w2,(i+1)*sizeof(int));
    }
    clock_t e = clock();
    printf("oa time = %f\n",(double)(e-s)/CLOCKS_PER_SEC);
    free(w1);
    free(w2);

}
```

oa time = 16.343000

Sumaryczna ilość przydzielonej pamięci dla jednego wskaźnika to $\frac{n(n-1)}{2} \approx 5 \cdot 10^{13}$ Tyle nie mam w komputerze…

Optymalizacja

```
int block_alloc(){
    int*w1=0;
    int*w2=0;
    int block_size=5;
    clock_t s = clock();
    for(int i=0;i<10000000;i++){
        if(i==block_size) {
            block_size*=2;
            w1 = realloc(w1, block_size * sizeof(int));
            w2 = realloc(w2, block_size * sizeof(int));
        }
    }
    clock_t e = clock();
    printf("ba time = %f\n",(double)(e-s)/CLOCKS_PER_SEC);
    free(w1);
    free(w2);
}
```

Przydzielamy pamięć w blokach. Rozmiar bloku rośnie razy 2 przy każdym przydziale.

ba time = 0.077000

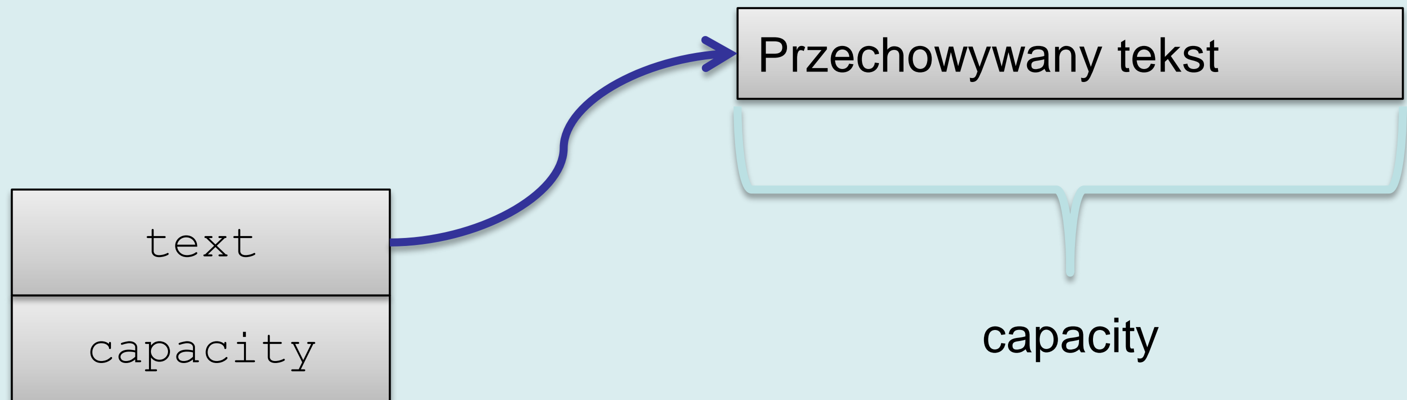
Czas wykonania około 200 tysięcy razy mniejszy!!!

Sumaryczna ilość przydzielonej pamięci dla jednego wskaźnika $\mathrm{Mem} = 10(1+2+4+\cdots 2^{n-1}) = 10\frac{1-2^n}{1-2}$. Spełnione jest $10 \cdot 2^{n-1} \geq 10^7$.

Logarytmując otrzymujemy $n = 21$, stąd $Mem = 20\,971\,510$

Przykład: String (1)

```
typedef struct tagString
{
        char*text;
        size_t capacity; // wielkosc bufora
}String;
```



Przykład: String (2)

```
/* inicjalizacja pól */
void init(String*string,const char*text)
{
        if(!text ){
                string->text=0;
                string->capacity=0;
        }else{
                string->text = strdup(text);
                string->capacity=strlen(text)+1;
        }
}
/* bezpieczny dostep do zawartosci */
const char*getText(const String*string)
{
        if(!string->text)return "";
        else return string->text;
}
```

Przykład: String (3)

```
/* dlugosc lancucha znakow */
size_t length(const String*string)
{
        if(!string->text)return 0;
        else return strlen(string->text);
}

/* zwolnienie pamieci */
void freeString(String*string)
{
        if(string->text)free(string->text);
        string->text=0;
        string->capacity=0;
}
```

Przykład: String (4)

```
/* przypisanie */
void set(String*string,const char*text)
{
        if(strlen(text)<string->capacity){
                strcpy(string->text,text);
                return;
        }
        freeString(string);
        init(string,text);
}

/* dodawanie tekstów */
String add(const String*a,const String*b)
{
        String result;
        result.text=(char*)malloc(length(a)+length(b)+1);
        strcpy(result.text,getText(a));
        strcat(result.text,getText(b));
        result.capacity = strlen(result.text)+1;
        return result;
}
```

Przykład: String (5)

```
/* rozszerzenie bufora */
int reserve(String*string,size_t newsize)
{
        if(newsize<string->capacity)return 0;
        if(string->text==0){
                string->text=(char*)malloc(newsize);
                (string->text)[0]=0;
        }else{
                string->text= (char*)realloc(string->text, newsize);
        }
        string->capacity=newsize;
        return string->capacity;
}

/* dopisanie znaku na końcu */
void append(String*string,int c)
{
    size_t len = length(string);
    if(string->capacity <= len+1)
        reserve(string, string->capacity ?2 * string->capacity:128);
    string->text[len]=c;
    string->text[len+1]=0;
}
```

Przykład: String (6)

```
int main()
{
        String a,b,c,d;
        /* dodawanie tekstów */
        init(&a,"Ala ma ");
        init(&b,"kota");
        c=add(&a,&b);
        printf("c=\"%s\"\n",getText(&c)); // c="Ala ma kota"
        freeString(&a);
        freeString(&b);
        freeString(&c);

        /* czytanie znakow , automatyczne powiekszanie bufora*/
        init(&d,0);  // pusty tekst
        for(;;){
                int ch;
                ch=getchar();
                if(ch==EOF)break;
                append(&d,ch);  // powiększa bufor do 128, 256, 512…
        }
        printf(getText(&d));
        freeString(&d);
        return 0;
}
```

Lista (1)

Lista jednokierunkowa przechowująca wartości całkowite.

Dwie struktury danych:

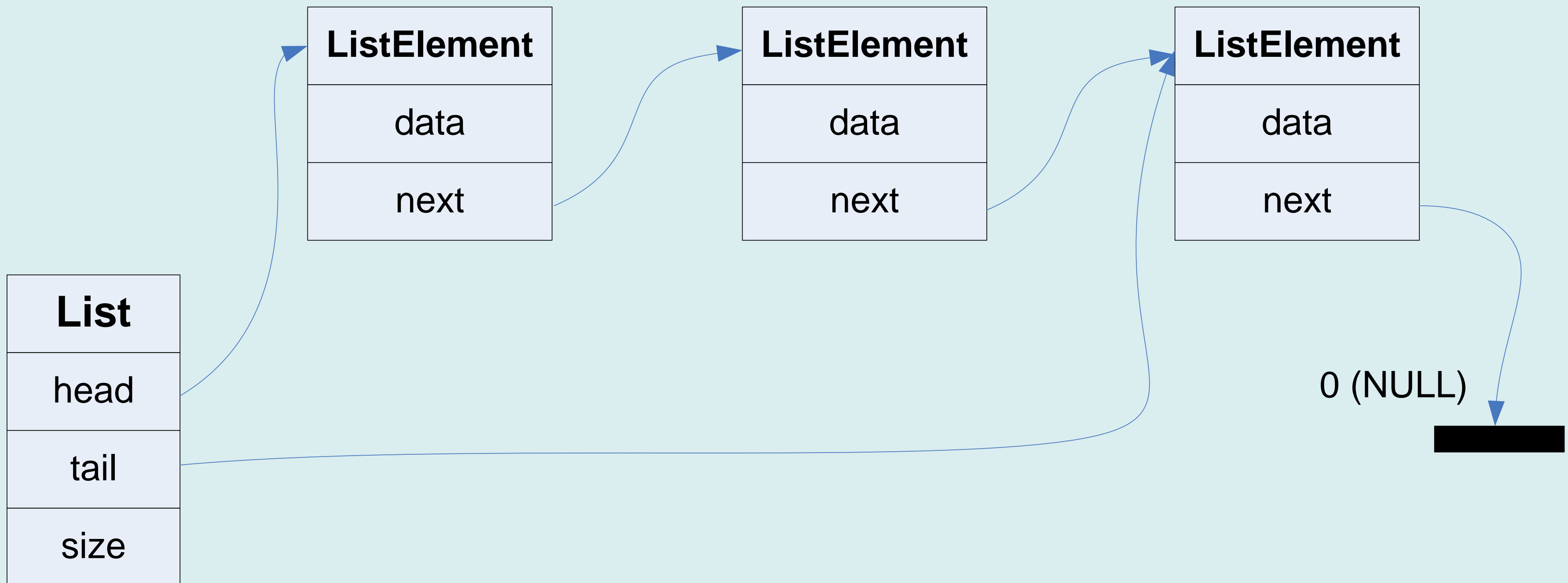
- `ListElement` (pomocnicza) – przechowuje dane
- `List` – wszystkie funkcje odwołują się do tej struktury

```
typedef struct tagListElement
{
        struct tagListElement*next;
        int data;
}ListElement;


typedef struct tagList
{
        ListElement*head;
        ListElement*tail;
        int size;
}List;
```

Lista (2)

Warunki spójności listy:

- Kiedy lista jest pusta, `head` i `tail` mają wartość 0 (NULL)
- Wskaźnik `next` ostatniego elementu listy ma wartość 0 (NULL)
- Pole `size` jest opcjonalne (może zostać obliczone poprzez iterację)



Lista (3)

```
/* inicjalizacja listy */
void init(List*list)
{
        list->head=0;
        list->tail=0;
        list->size=0;
}

/* Dodawanie danych do listy */
void push_front(List*list, int data)
{
        ListElement*element =
        (ListElement*)malloc(sizeof(ListElement));
        element->next=list->head;
        element->data=data;
        if(list->head!=0){
                list->head=element;
        }else{
                list->head=list->tail=element;
        }
        list->size++;
}
```

Lista (4)

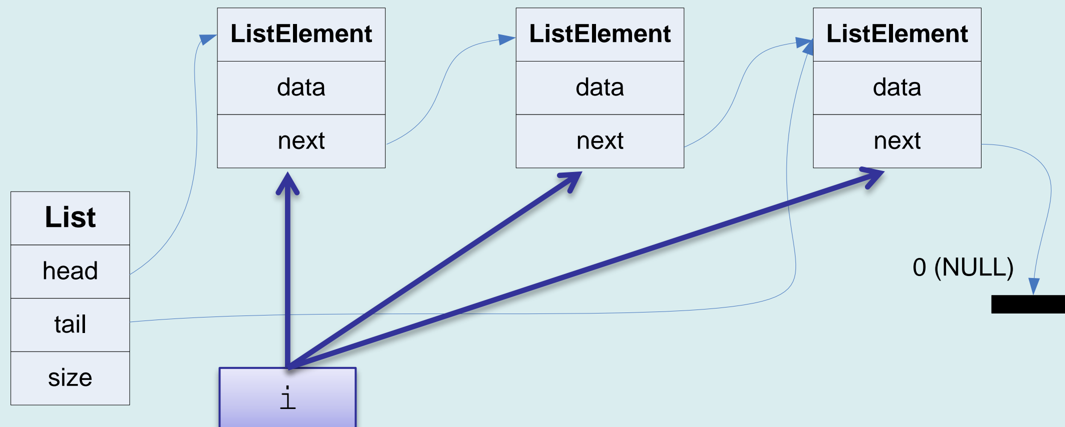
```
void push_back(List*list, int data)
{
        ListElement*element =
         (ListElement*)malloc(sizeof(ListElement));
        element->next=0;
        element->data=data;
        if(list->tail!=0){
                list->tail->next=element;
                list->tail=element;
        }else{
                list->head=list->tail=element;
        }
        list->size++;
}
```

Lista (5)

```
/* Usuwanie pierwszego elementu */

void delete_front(List*list){
     ListElement*toDelete;
     if(list->head==0)return;
     toDelete = list->head;
     list->head=list->head->next;
     if(list->head==0)list->tail=0;
     free(toDelete);
     list->size--;
}

/* Zwalnianie całej listy */

void freeList(List*list)
{
       while(list->head){
              delete_front(list);
       }
       printf("\nTRACE: stan listy %p %p %d\n",
              list->head, list->tail, list->size);
}
```

Lista (6)

```
/* Wypisanie zawartości listy, iteracja po elementach listy */

void dumpList(const List*list)
{
        ListElement*i;
        for(i=list->head; i!=0; i=i->next){
                printf("%d ",i->data);
        }
        printf("\n");
}
```



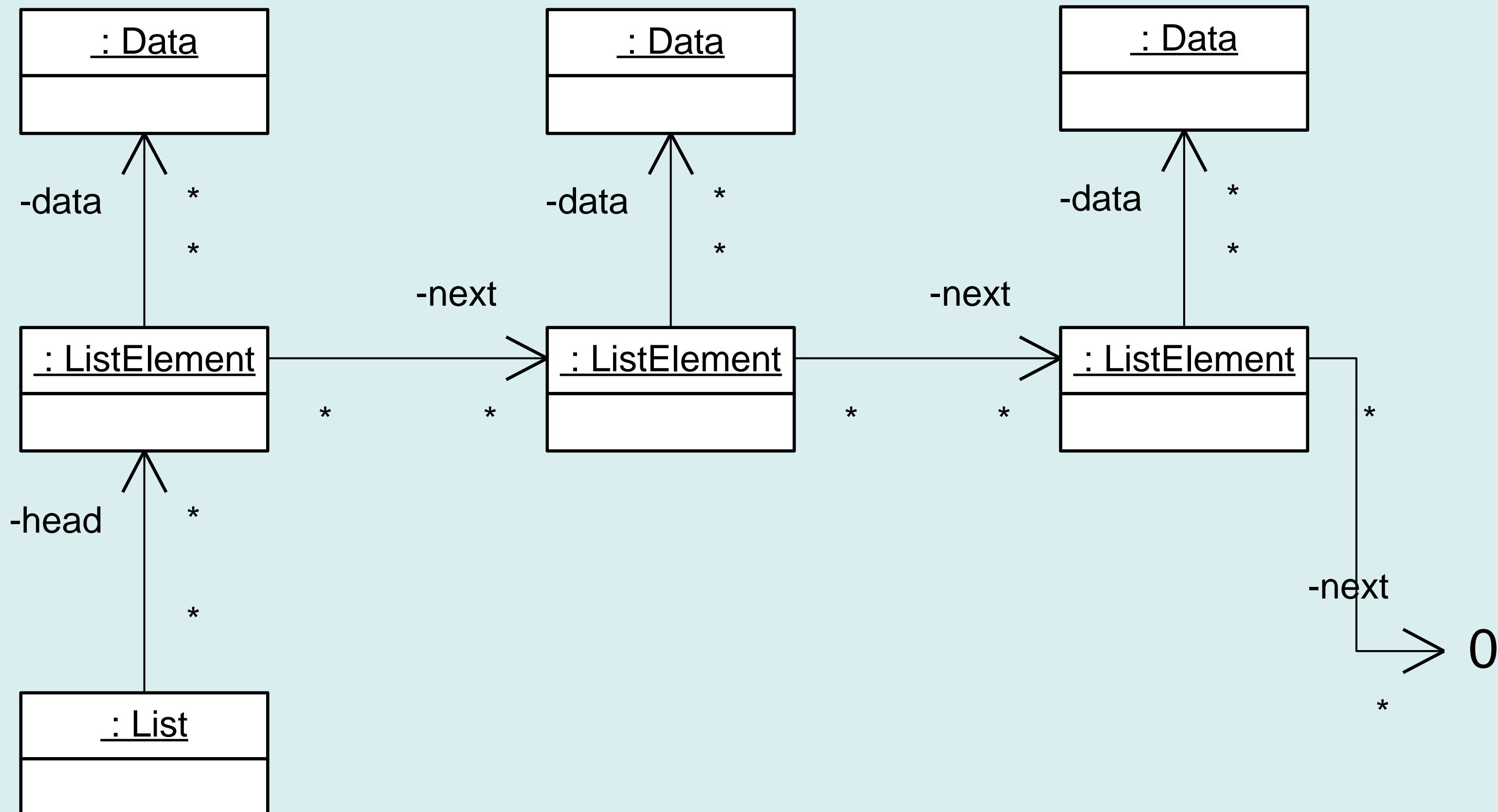
Lista (5)

```
int main(int argc, char *argv[])
{
  int i;
  List list;
  init(&list);
  for(i=0;i<10;i++){
      push_front(&list,i);
      push_back(&list,i);
  }
  dumpList(&list);

  for(i=100;i<105;i++){
      delete_front(&list);
      push_back(&list,i);
  }
  dumpList(&list);

  freeList(&list);
  return 0;
}
```

```
9 8 7 6 5 4 3 2 1 0 0 1 2 3 4 5 6 7 8 9
4 3 2 1 0 0 1 2 3 4 5 6 7 8 9 100 101 102 103 104

TRACE: stan listy 00000000 00000000 0
```

Lista ogólnego przeznaczenia (1)

```
typedef struct tagListElement
{
        struct tagListElement*next;
        void*data;
}ListElement;
```



Lista ogólnego przeznaczenia (2)

```
typedef void (*ConstDataFp)(const void*);
typedef void (*DataFp)(void*);
typedef int (*CompareDataFp)(const void*,const void*);


typedef struct tagList
{
        ListElement*head;
        int size;
        ConstDataFp dumpData;
        DataFp freeData;
        CompareDataFp compareData;
}List;
```

Możliwości adaptacji:

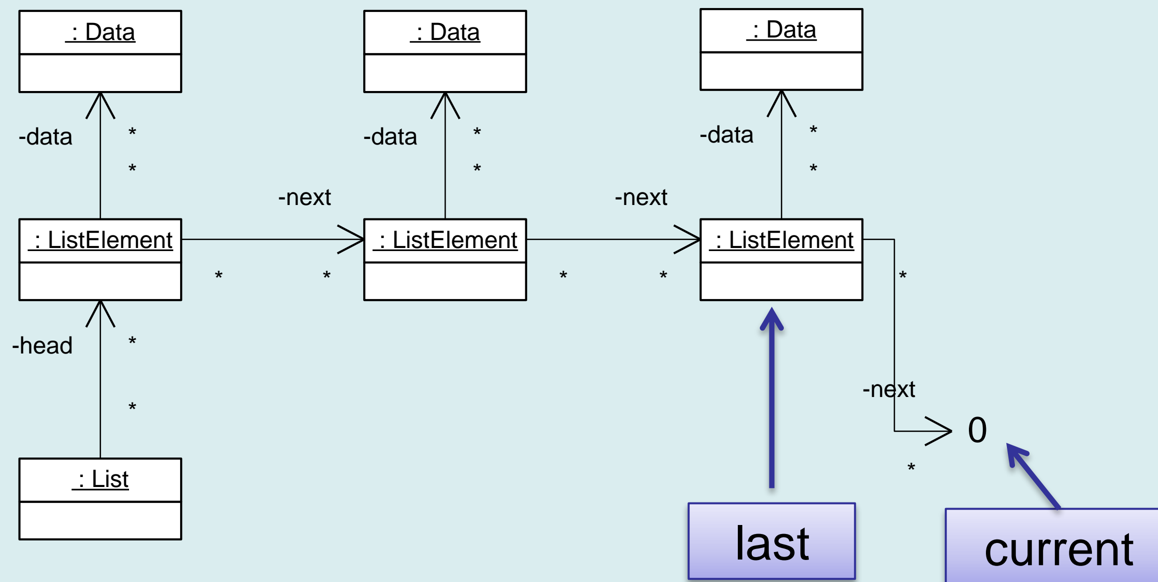
- Funkcja **`dumpData`** – potrafi zinterpretować i wypisać dane
- Funkcja **`freeData`** – jeżeli pamięć dla danych jest przydzielana dynamicznie, odpowiada za jej zwalnianie podczas usuwania elementów listy
- Funkcja **`compareData`** – porównuje dane podczas wstawiania, umożliwia to sortowanie elementów według różnych kryteriów

Lista ogólnego przeznaczenia (3)

```
/* inicjalizacja listy */
void init(List*list) {
        list->head=0;
        list->size=0;
        list->dumpData=0;
        list->freeData=0;
        list->compareData=0;
}

 /* zwolnienie pamięci listy */
void freeList(List*list) {
        ListElement*current = 0;
        current = list->head;
        while(current!=0){
                ListElement*todelete = current;
                current=current->next;
                if(list->freeData)list->freeData(todelete->data);
                free(todelete);
        }
        list->size=0;
        list->head=0;
}
```

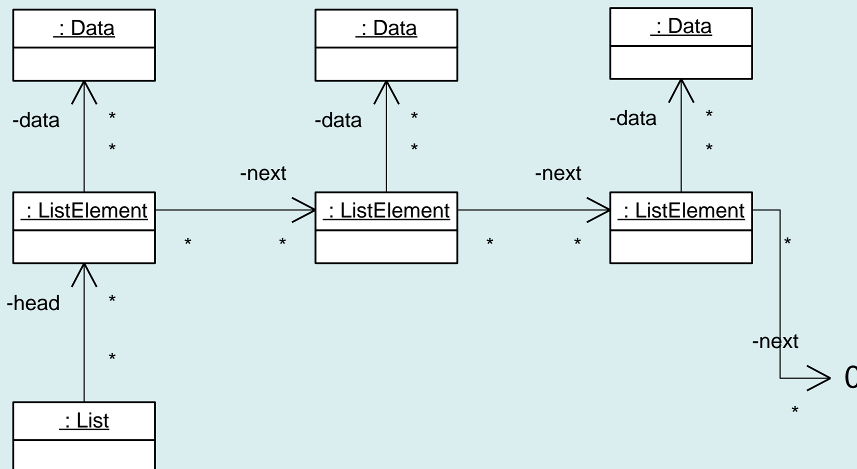
Lista ogólnego przeznaczenia (4)

```
/* poszukiwanie ostatniego elementu */
ListElement*findLast(const List*list)
{
 ListElement*last = 0;
 ListElement*current = 0;
 for(current = list->head;current!=0;current=current->next){
       last = current;
 }
 return last;
}
```



Lista ogólnego przeznaczenia (5)

```
/* miejsce wstawienia */
ListElement*findInsertionPoint(const List*list, ListElement* element)
{
 ListElement*insertionPoint = 0;
 ListElement*current = 0;
 for(current = list->head;current!=0;current=current->next){
     if( list->compareData(current->data,element->data)<=0)
         insertionPoint=current;
     }  // else break;
 return insertionPoint;
}
```



- Funkcja znajduje element **po którym** należy wstawić nowy element.
- Jeżeli zwróci 0, oznacza to, że element należy dodać na początku listy…

Lista ogólnego przeznaczenia (6)

```
/* dodanie danych do listy */
void add(List*list,void*data)
{
 ListElement*element = (ListElement*)malloc(sizeof(ListElement));
 element->next=0;
 element->data=data;

 if(!list->compareData){   // bez sortowania
     ListElement*last = findLast(list);
     if(last==0)list->head=element;
     else last->next=element;
 }else{                   // sortowanie podczas wstwiania
     ListElement*insertionPt = findInsertionPoint(list,element);
     if(insertionPt==0){
         element->next=list->head;
         list->head=element;
     }else{
         element->next=insertionPt->next;
         insertionPt->next=element;
     }
 }
 list->size++;
}
```

Lista ogólnego przeznaczenia (7)

```
/* wypisanie zawartości */
void dumpList(const List*list)
{
        ListElement*i;
        for(i=list->head;i!=0;i=i->next){
                if(list->dumpData)list->dumpData(i->data);
                else printf("%p ",i->data);

        }
}

/* testy */
void printString(const void*data)
{
        printf("%s ",data);
}
```

Lista ogólnego przeznaczenia (8)

```
void test1()
{
        List list;
        init(&list);
        list.dumpData=printString;

        add(&list,"Ala");
        add(&list,"ma");
        add(&list,"kota");
        add(&list,"i");
        add(&list,"psa");
        dumpList(&list);
        freeList(&list);
}
```

```
Ala ma kota i psa
```

Lista przechowuje wskaźniki do stałych tekstowych (pamięć dla nich nie jest przydzielana na stercie) i nie ma potrzeby ich zwalniać.

Lista ogólnego przeznaczenia (8)

```
int compareString(const void*e1,const void*e2)
{
	return stricmp((const char*)e1,(const char*)e2);
}

void test2()
{
	List list;
	init(&list);
	list.dumpData=printString;
	list.compareData=compareString;

	add(&list,"Ala");
	add(&list,"ma");
	add(&list,"kota");
	add(&list,"i");
	add(&list,"psa");
	dumpList(&list);
	freeList(&list);
}
```

```
Ala i kota ma psa
```

Lista ogólnego przeznaczenia (8)

```
#include <time.h>
void freeMemory(void*el)
{
        free(el);
}

void test3()
{
        List list;
        init(&list);
        list.dumpData=printString;
        list.compareData=compareString;
        list.freeData=freeMemory;
        int i;
        srand( (unsigned)time( NULL ) );
        for( i = 0;   i < 100;i++ ){
                char buf[256];
                sprintf(buf,"%d",rand());
                add(&list,strdup(buf));
        }
        dumpList(&list);
        freeList(&list);
}
```

```
10068 10131 10300 11624 11803 12371 12517 12554 13044 13059
1314 13394 14875 14919 14932 14934 15442 15451 15654 1590
16675 16702 16840 1757 17629 17724 18031 18157 18293 18318
18771 18982 1924 19279 19648 19812 20144 20650 20807 21164
21521 22446 22602 22815 23483 23601 238 24203 24291 25351
25388 25541 25826 25930 26014 26463 26721 27260 27653 28973
29076 29159 29342 29472 2956 29919 30390 3090 31044 3137
3195 32267 32525 3328 3337 337 3491 3881 3950 4245 4464
4539 4698 5077 5414 5451 559 6364 6766 6969 7792 7839 7946
8044 8089 811 8847 9130 9346 983
```

Gdyby nie było `strdup()`, wskaźniki w liście wskazywałyby na ten sam element `buf`

Co należy zapamiętać

- Funkcje malloc() i free()
- Alokacja tablic
- Struktury typu tablica tablic
- Implementacje list