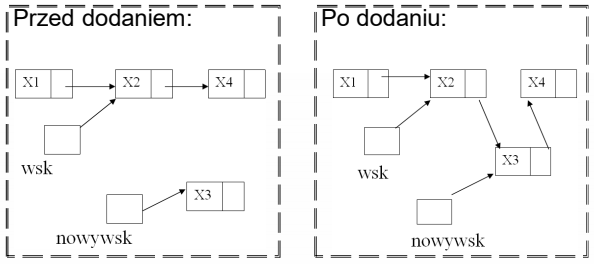
ISO/ANSI C – zm. dynamiczne

Dodanie do listy uporządkowanej (np.wg roku)



147

ISO/ANSI C – zm. dynamiczne

Dodanie do listy uporządkowanej malejąco (np. wg roku)

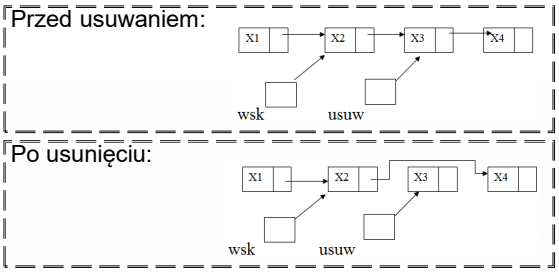
```
nowywsk = malloc(sizeof(struct film_t));
strcpy(nowywsk->tytul,"Bogowie");
nowywsk->rok=2014;
nowywsk->nast = NULL;

if(glowa==NULL)
    glowa = wsk = nowywsk;
else
if (glowa->rok < nowywsk->rok) {
    nowywsk->nast = glowa;
    glowa = nowywsk;
}
```

```
else
    if (glowa->nast == NULL)
        glowa->nast = nowywsk;
else {
    wsk = glowa;
    while (wsk->nast->rok > nowywsk->rok) {
        wsk = wsk->nast;
        if (wsk->nast == NULL)
            break;
    };
    nowywsk->nast = wsk->nast;
    wsk->nast = nowywsk;
}
```


148

ISO/ANSI C – zm. dynamiczne

Usuwanie wskazanego elementu z listy



149

ISO/ANSI C – zm. dynamiczne

Usuwanie wskazanego elementu z listy *(filmu z roku 2010)*

```
rok = 2010; // wskazuje na rok
            // filmu do usunięcia

if (glowa->rok == rok) {
    usuw = glowa;
    glowa = usuw->nast;
}
```

```
else {
    wsk = glowa;
    while ((wsk->nast != NULL)) {
        if (wsk->nast->rok == rok)
            break;
        wsk = wsk->nast;
    };
    if(wsk->nast != NULL) {   // znaleziono!
        usuw = wsk->nast;
        wsk->nast = wsk->nast->nast;
        free(usuw);
    };
}
```


150

ISO/ANSI C – zm. dynamiczne

Zaleta list jednokierunkowych:

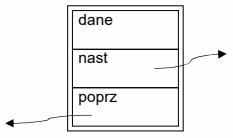
- zużywają pamięć proporcjonalnie do potrzeb

Wada list jednokierunkowych:

- wskaźniki mogą przesuwać się jedynie do przodu

Listy dwukierunkowe:

```
struct film_t {
    char tytul[80];
    int rok;
    struct film_t *nast;
    struct film_t *poprz;
};
```

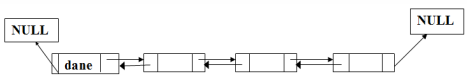


151

ISO/ANSI C – zm. dynamiczne

Listy dwukierunkowe:

- możliwość poruszania się wskaźnika w obydwu kierunkach
- zużywają minimalnie więcej pamięci, są za to szybsze w dostępie do danych

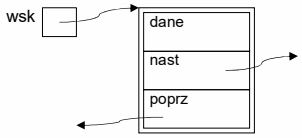


152

ISO/ANSI C – zm. dynamiczne

Listy dwukierunkowe:

```
struct film_t *wsk;
..
wsk = wsk->nast;  /* Przemieszczanie się w przód: */
..
wsk = wsk->poprz; /* Przemieszczanie się w tył: */
```



153

ISO/ANSI C – zm. dynamiczne

Listy dwukierunkowe – dodawanie:

`struct film_t *wsk, *nowy;`

nowy – wskazuje na nowy element do umieszczenia

wsk – wskazanie na element po którym mamy umieścić 'nowy'

```
nowy->nast = wsk->nast;
wsk->nast->poprz = nowy;
wsk->nast = nowy;
nowy->poprz = wsk;
```


154

ISO/ANSI C – zm. dynamiczne

Listy dwukierunkowe – dodawanie:

`struct film_t *wsk, *nowy;`

nowy – wskazuje na nowy element do umieszczenia

wsk – wskazanie na element przed którym mamy umieścić 'nowy'

```
nowy->nast = wsk;
nowy->poprz = wsk->poprz;
wsk->poprz->nast = nowy;
wsk->poprz = nowy;
```


155

ISO/ANSI C – zm. dynamiczne

Listy dwukierunkowe – usuwanie:

`struct film_t *wsk, *nowy;`

wsk – wskazanie na element, po którym znajduje się element do usunięcia

usun – wskaźnik pomocniczy

```
usun = wsk->nast;
wsk->nast = wsk->nast->nast;
wsk->nast->poprz = wsk;
free(usun);
```


156

ISO/ANSI C – zm. dynamiczne

Listy dwukierunkowe – usuwanie:

`struct film_t *wsk, *nowy;`

wsk – wskazanie na element, przed którym znajduje się element do usunięcia

usun – wskaźnik pomocniczy

```
usun = wsk->poprz;
wsk->poprz = wsk->poprz->poprz;
wsk->poprz->nast = wsk;
free(usun);
```


157

ISO/ANSI C – zm. dynamiczne

Wnioski:

- Listy są przydatne szczególnie wtedy, gdy:
  - Dodawane są lub usuwane dowolne elementy ciągu, oraz
  - nie jest znana ich liczba.
- Tablice i pliki również umożliwiają wykonanie tych operacji, ale..
  .. dla tablicy musimy przepisywać elementy leżące za wstawianym lub usuwanym elementem, a dla plików musimy przepisywać całe pliki.
- Tablice są wygodniejsze do realizacji operacji wyszukiwania, przeglądania i dostępu do wybranego elementu. (dla tablicy posortowanej oszacowanie z góry kosztu tzw. wyszukiwania binarnego to *log(n)*, natomiast dla list stosowane jest wyszukiwanie liniowe – *n*).

158

159

ISO/ANSI C – typy pochodne

Najbardziej nawet złożone struktury mają swój typ, który można zapisać za pomocą istniejących podstawowych typów danych

Przykład:

```
int tab[] = {1,2,3};
int *y[3] = { tab, tab, tab };
int (*z)[3] = &tab;
```

Takim złożonym typom można nadać *aliasy* (nazwy) za pomocą deklaracji **typedef**

Przykład:

```
typedef int T[];
typedef int *T3Wsk[3];
typedef int (*WskT3)[3];

T tab = {1,2,3};
T3Wsk tab1 = {tab, tab, tab };
WskT3 tab2 = &tab;
```


160

161

ISO/ANSI C – obsługa błędów

Poprawa obsługi błędów to jedna z najpewniejszych metod tworzenia solidnie działającego kodu

Źródła błędów w działających programach – sytuacje niezgodne z oczekiwaniami autora programu, ale potencjalnie możliwe do wystąpienia

Przyczyna:

fałszywe oczekiwania autora programu wynikające z lenistwa, albo z „wishful thinking" czyli pobożnych życzeń autora, że wszystkie warunki prawidłowej pracy zostaną spełnione przez użytkownika, zanim uruchomi on program, albo..

162

ISO/ANSI C – obsługa błędów

Obsługa błędów w C polega na kontrolowaniu rezultatów wykonania wywoływanych funkcji i powiązaniu z tymi rezultatami kodu obsługującego błąd.

Przykład:

Otwieranie pliku (*może się udać, lub nie*).

```
FILE* stream;
char s[100];
…
stream  = fopen( nazwa_pliku, "r" ));
fscanf(stream, "%s", s);
```

Czy ten kod jest bezpieczny?

163

ISO/ANSI C – obsługa błędów

Przykład:

```
FILE* stream;
char s[100], file_name[100];
…
stream  = fopen( nazwa_pliku, "r" ));
if (stream == NULL) /* sprawdzamy, czy otwieranie się powiodło */
        ...
```

Co powinno się znaleźć w miejscu kropeczek?

1. Przerwanie pracy całego programu
2. Wyświetlenie komunikatu o problemie i powrót sterowania do początku działania programu
3. Powrót sterowania do miejsca, w którym pobierana jest nazwa pliku

164

ISO/ANSI C – obsługa błędów

Podstawowa strategia obsługi błędów – sprawdzanie wartości zwracanej przez funkcję biblioteczną, której wywołaniu mogło towarzyszyć pojawienie się błędu

Jeżeli informacje zwracane przez funkcję biblioteczną są zbyt proste, dodatkowo wykorzystywana jest zmienna globalna przechowująca kod błędu: **`errno`** z biblioteki **`<errno.h>`**

Zmienna **`errno`** ma ustawiany kod błędu w momencie wywołania żądania systemowego, takiego jak np. próba otwarcia pliku.

Należy ja od razu odczytać, bo następne wywołanie może spowodować kolejną zmianę jej wartości.

Mając kod błędu możemy zażądać tekstu odpowiadającego temu kodowi i w ten sposób dowiedzieć się czegoś więcej

165

ISO/ANSI C – obsługa błędów

Constant	System error message	Value
E2BIG	Argument list too long	7
EACCES	Permission denied	13
EAGAIN	No more processes or not enough memory or maximum nesting level reached	11
EBADF	Bad file number	9
ECHILD	No spawned processes	10
EDEADLOCK	Resource deadlock would occur	36
EDOM	Math argument	33
EEXIST	File exists	17
EINVAL	Invalid argument	22
EMFILE	Too many open files	24
ENOENT	No such file or directory	2
ENOEXEC	Exec format error	8
ENOMEM	Not enough memory	12
ENOSPC	No space left on device	28
ERANGE	Result too large	34
EXDEV	Cross-device link	18

166

ISO/ANSI C – obsługa błędów

Jeżeli chcemy wyświetlić użytkownikowi komunikat z tekstem odpowiadającym kodowi błędu:

```
char *strerror( int errnum );
```

z biblioteki **`<string.h>`**

Funkcja zwraca tekst, który odpowiada opisowi kodu błędu.

```
file = fopen(fname, "r");
if (file == NULL) {
  printf("Error while trying to open '%s': %s\n",
                                  fname, strerror(errno));
  ….
```


167