Dynamiczne przydzielanie pamięci

1. Wstęp teoretyczny

Pamięć strukturom danych (obiektom) jest przydzielana:

- statycznie (w trakcie kompilacji kodu)
- dynamicznie (w trakcie działania programu)

Przydział dynamiczny następuje:

- automatycznie (dla obiektów lokalnych)
- „na żądanie" programisty (przydzielenie poprzez pamięci dla wskaźników)

Do stworzenia obiektu w sposób dynamiczny „na żądanie" konieczne jest posiadanie wskaźnika na typ obiektu i mechanizmu przydzielania pamięci temu obiektowi. Wskaźnik deklaruje programista, natomiast mechanizm jest zawarty w bibliotece standardowej. **Tak więc przydzielanie pamięci nie jest częścią języka C – jest to „dodatek" udostępniony przez bibliotekę standardową.** Z punktu widzenia programisty mechanizm przydzielania pamięci jest widziany jako trzy funkcje przydzielające pamięć i jedna funkcja zwalniająca niepotrzebną pamięć. Te cztery funkcje są de facto interfejsem mechanizmu przydzielania pamięci, natomiast jego implementacja jest ukryta w bibliotece standardowej (nie będziemy się zajmować implementacją – zwłaszcza, że istnieją tu różne rozwiązania w różnych kompilatorach).

Poniższe deklaracje funkcji do dynamicznego zarządzania pamięcią są umieszczone w pliku „stdlib.h" (mogą też być umieszczone w „alloc.h" lub „mem.h").

1.1 Funkcje zarządzania pamięcią

Funkcje przydzielające pamięć:

```
void * malloc(size_t rozmiar);
void * calloc(size_t liczba_elementow,  size_t rozmiar_elementu);
void * realloc(void * ptr, size_t nowy_rozmiar);
```

gdzie: size_t - typ wyniku operatora sizeof (najczęściej identyczny z unsigned int)

Funkcje **malloc** i **calloc** tylko przydzielają pamięć natomiast **realloc** zwalnia pamięć wcześniej przydzieloną przez **malloc** lub **calloc** i przydziela ją na nowo (najczęściej chodzi tu o przydzielenie innej ilości pamięci).

Różnica pomiędzy **malloc** a **calloc**:

calloc zeruje przydzielaną pamięć, **malloc** nie zeruje (przydzielona pamięć zawiera „śmieci").

realloc przydziela pamięć bez zerowania (tak jak **malloc**).

Jeżeli w funkcji realloc nowa ilość pamięci jest większa lub równa starej, to „stara" jej zawartość jest przepisywana w całości do nowoprzydzielonej. W przeciwnym wypadku (nowa mniejsza od starej) przepisywane jest tyle, ile się zmieści (czyli następuje obcięcie części danych).

Funkcja zwalniająca pamięć:

```
void free(void *ptr);
```

Funkcja **free** jako parametr dostaje wskaźnik, którego wartość jest zwracana przez **malloc**, **calloc** lub **realloc**.

Funkcje **malloc**, **calloc** i **realloc** zwracają wskaźnik do (inaczej adres) przydzielonej pamięci lub **NULL** w przypadku błędu (np. brak pamięci do przydzielenia).

Funkcja **free** nic nie zwraca.

Wywołanie: **realloc(NULL, rozmiar)** odpowiada wywołaniu: **malloc(rozmiar)**

Wywołanie: **realloc(ptr, 0)** odpowiada wywołaniu: **free(ptr)**

Uwagi:

PO PRZYDZIELENIU PAMIĘCI NALEŻY SPRAWDZIĆ, CZY FUNKCJA PRZYDZIELAJĄCA NIE ZWRÓCIŁA NULL!

PRZYDZIELONĄ PAMIĘĆ NALEŻY ZAWSZE ZWALNIAĆ!

Dygresja:

Po wywołaniu:

```
free(ptr);
```

wartość parametru **ptr** nie ulega zmianie! Oznacz to, że **ptr** dalej wskazuje na pewne miejsce pamięci, ale ta wskazywana pamięć już nie jest nigdzie przydzielona. Tak więc użycie **ptr** po zwolnieniu pamięci (inne niż ponowny jej przydział) spowoduje nieprzewidywalne zachowanie programu. Co gorsza taki błąd może nie zostać od razu dostrzeżony. Program może nawet czasami działać poprawnie. Dlatego dobrze jest dodać po **free(ptr)** linię:

```
ptr=NULL;
```

Dzięki temu każde następne nieprawidłowe użycie **ptr** spowoduje przerwanie pracy programu i co da możliwość wychwycenia i naprawy błędu.

2. Podsumowanie

W języku C program dysponuje następującymi rodzajami pamięci:

- pamięć statyczna – przechowuje obiekty (w znaczeniu stałe lub zmienne – nie w sensie programowania obiektowego) tworzone w trakcie kompilacji programu.
- stos - przechowuje obiekty lokalne (tworzone „automatycznie” w trakcie działania programu).
- sterta - przechowuje obiekty tworzone dynamicznie w trakcie działania programu.

Zalety dynamicznej alokacji pamięci:

- ilość potrzebnej pamięci może zostać wyliczona w trakcie pracy programu
- nieużywana pamięć może zostać zwolniona i powtórnie wykorzystana na inne potrzeby

Wada dynamicznej alokacji pamięci:

- **programista jest odpowiedzialny za kontrolę przydziału i zwalniania pamięci**

3. Przykład dynamicznej alokacji pamięci

```
#include „stdio.h”
#include „stdlib.h”

int * fun(int n)
{
int *ptr;
int i;

if(n<2) return NULL;

ptr = malloc(n * sizeof(int));
if(ptr == NULL) return NULL;

ptr[0]=ptr[1]=1;
for(i=2; i<n; i++)     ptr[i]=ptr[i-1]+ptr[i-2];

return ptr;
}


main()
{
int *tab;
int i;

tab = fun(25);
if(tab != NULL)
  {
     for(i=0; i<25; i++)     printf(„%d \n”, tab[i]);

     free(tab);
     tab = NULL;
  }
}
```

Co robi powyższy program?

ZADANIE 1

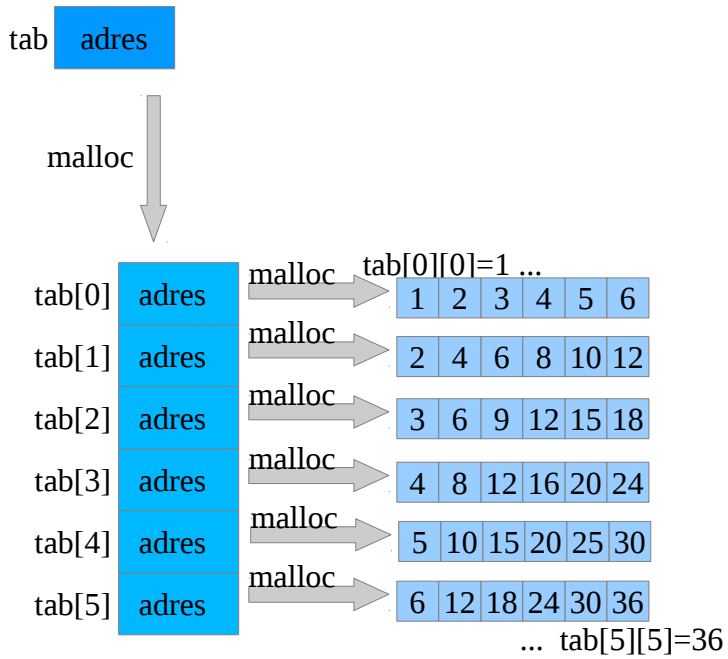
Napisz program, który będzie zawierał funkcję do dynamicznego tworzenia dwuwymiarowej, kwadratowej tablicy liczb całkowitych o wymiarze nxn. Funkcja powinna otrzymywać rozmiar tablicy (n) i zwracać odpowiedni wskaźnik (podobnie jak w przykładzie).

Przykładowa deklaracja takiej funkcji:

```
int ** make_tab_2D( int n );
```

W funkcji należy najpierw przydzielić pamięć na tablicę wskaźników, a potem każdemu wskaźnikowi w tablicy przydzielić wiersz tablicy liczb całkowitych (jak na rysunku).

```
int ** tab;
```



Rys 1. Utworzenie dynamicznie przydzielanej tablicy o rozmiarze 6x6 (następnie wypełnionej fragmentem „tabliczki mnożenia”)

Napisaną funkcję proszę wykorzystać w programie main do utworzenia tablicy, którą należy potem wypełnić „tabliczką mnożenia” w zakresie 10x10.

Szkielet programu:

```
#include „stdlib.h”

int ** make_tab_2D( int n )
{
TU PROSZĘ WPISAĆ KOD
}

main()
{
int **tab;
int i;

tab = make_tab_2D(10);
if(tab != NULL)
  {
     TU PROSZĘ WPISAĆ KOD

     free(tab);
     tab = NULL;
  }
}
```

ZADANIE 2

Zmodyfikuj program z zajęć „Struktury”, tak aby zamiast tablicy struktur stworzyć listę jednokierunkową struktur. W tym celu należy dodać do struktury wskaźnik „na siebie samą”.

```
struct student {
char Nazwisko[30];
char Imie[15];
short czyMaZaleglosci;
float SredniaOcen;
struct student *Nast;
};
```

Zmodyfikowany początek programu powinien wyglądać tak:

```
#include <stdio.h>
#include <stdlib.h>
#include "studenci.h"

struct student *Studenci=NULL;

main() {
```

… reszta tak jak w konspekcie „Struktury”

Napisz nowe wersje funkcji:

```
void WypiszListeStudentow();
void DodajOpisStudenta();
```

oraz dopisz nową funkcję usuwającą wszystkich studentów z listy:

```
void UsunListeStudentow();
```

(funkcje void ModyfikujOpisStudenta(); i void UsunOpisStudenta(); nie będą używane, nie trzeba ich pisać, można zakomentować kod, który się do nich odnosił)

Dodawanie nowego studenta powinno zawierać (m.in.) następujący kod:

```
struct student *nowy_student;

nowy_student = malloc( sizeof( struct student ) );
if(nowy_student != NULL)
{
     nowy_student->Nast = Studenci;
     Studenci = nowy_student;
}
```

Wypisywanie listy powinno zawierać (m.in.) następujący kod:

```
struct student *aktualny = Studenci;

while(aktualny != NULL)
{
     TU PROSZĘ WPISAĆ KOD
     aktualny = aktualny->Nast;
}
```