

1 Dynamiczne struktury danych (LAB 1)

1.1 Wprowadzenie

W praktyce programistycznej bardzo często znajdują zastosowanie rekursywne struktury danych. Typowymi ich przykładami są listy jedno- i dwukierunkowe, stos oraz drzewa. Konstrukcja takich obiektów wymaga znajomości złożonych typów danych oraz technik dynamicznego przydziału pamięci. Tematem niniejszego opracowania jest omówienie mechanizmów języka C pozwalających na konstruowanie struktur rekursywnych.

1.2 Proces kompilacji kodu

Procedura kompilacji i konsolidacji programu w języku C składa się najczęściej z czterech etapów:

1. **Prekompilacja (preprocessing).** Wstępny etap, na którym przetwarzane są instrukcje preprocesora związane z dołączaniem plików, komplikacją warunkową czy makrami.
2. **Kompilacja (compilation).** Kod przetworzony przez preprocesor zostaje przekształcony na kod źródłowy w języku assemblera.
3. **Assemblacja (assembly).** Na tym etapie budowane są pliki obiektów (pliki półskompilowane), tworzone z kodu assemblera przetworzonego na język maszynowy.
4. **Linkowanie/konsolidacja (linking).** W ostatnim kroku pliki obiektów oraz (ewentualnie) bibliotek łączone są, w wyniku czego powstaje pojedynczy plik wykonywalny. W wynikowym pliku wykonywalnym mogą znajdować się odwołania do elementów zdefiniowanych w innych plikach (o tym więcej podczas kolejnych zajęć laboratoryjnych).

Dostępnych jest wiele różnych kompilatorów języka C realizujących powyższe zadania. Do najpopularniejszych należą, dostępne typowo w systemach UNIX/Linux lub pod Windows z wykorzystaniem minGW lub MSYS2, `gcc` oraz `clang`. W systemie Windows w środowisku Microsoft Visual C++ dostępny jest kompilator `MSVC`.

Większość zadań laboratoryjnych wykorzystywać będzie kompilator `gcc`, stąd przypomnijmy podstawową składnię wywołania kompilatora.

Załóżmy, że mamy plik źródłowy `program.c`, który chcielibyśmy poddać pełnemu procesowi kompilacji i linkowania za pomocą kompilatora `gcc` (Linux). Robimy to za pomocą poniższego polecenia:

```
$ gcc -Wall program.c
```

Kompilator domyślnie stworzy plik wykonywalny o nazwie `a.out`, umieszczony w bieżącym folderze. Uruchomienie tego pliku:

```
$ ./a.out
```

UWAGA: opcja kompilacji `-Wall` służy do włączenia wyświetlania informacji o ostrzeżeniach (warnings) podczas kompilacji. Chociaż ostrzeżenia nie blokują (jak błędy kompilacji) możliwości uruchomienia pliku wykonywalnego, to jednak dobrą praktyką programistyczną jest przejrzenie listy zgłoszonych ostrzeżeń i próba modyfikacji kodu tak, aby je zlikwidować. Z tego powodu należy zawsze korzystać z opcji `-Wall`!

Jeśli chcemy zmienić nazwę tworzonego pliku wykonywalnego (ew. również jego lokację) korzystamy z dodatkowej opcji `-o`. Poniżej przykład wywołania kompilacji z tą opcją oraz uruchomienia pliku wynikowego:

```
$ gcc -o program -Wall program.c
$ ./program
```

1.3 Dynamiczna alokacja pamięci

Dynamiczna alokacja pamięci polega na rezerwowaniu obszarów pamięci operacyjnej na zadanie w trakcie działania programu. Mechanizm ten stosowany jest do tworzenia struktur danych o nieznanej z góry wielkości. Język C oferuje grupę kilku funkcji dynamicznego przydziału i zwalniania pamięci. Funkcje przydziału pamięci zwracają tzw. wskaźnik uniwersalny typu `void*`. Jest to wskaźnik na dowolny typ. Dobra praktyka programistyczna wymaga jawnego rzutowania go na typ stosowanego obiektu. Zostanie to później omówione na przykładzie. Prototypy omawianych niżej funkcji znajdują się w systemowym pliku nagłówkowym `stdlib.h`. Podstawową funkcja dynamicznego przydziału pamięci jest `malloc()`. Jej prototyp jest następujący:

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

Parametrem funkcji jest liczba rezerwowanych bajtów. W przypadku błędu zwracana jest wartość `NULL`. Przydzielona pamięć nie jest zerowana. Obok opisanej wyżej funkcji, która nie inicjalizuje rezerwowanego obszaru, dostępna jest `calloc()` zerująca przydzieloną pamięć:

```
void* calloc(size_t noitems, size_t size)
```

Funkcja alokuje bloki (`noitems * size`) bajtów i wypełnia je zerami. Zwalnianiu dynamicznie rezerwowanych obszarów służy funkcja `free()`. Jej prototyp jest następujący:

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

Przykład użycia funkcji `malloc()` i `free()`:

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

void main ( void )
{
    char* lancuch ;
    lancuch = (char*) malloc(10); /* przydziel pamiec */
    strcpy (lancuch , " HELLO "); /* skopiuj napis */
    printf ("\nLancuch zawiera napis : %s", lancuch);
    free (lancuch); /* zwolnij przydzielona pamiec */
}
```

Uwaga! Przy alokacji pamięci zastosowane zostało jawnie rzutowanie typu `void*` na typ wskaźnika używanego obiektu, czyli `char*`.

1.4 Wyłapywanie przecieków pamięci

Alokacja i dealokacja (zwalnianie) pamięci przez programistę niesie ze sobą ryzyko pojawienia się **przecieków pamięci** w sytuacji gdy operacje te nie zostaną wykonane w pełni poprawnie (np. dla zaalokowanego bloku pamięci przez `malloc()` nie nastąpi jego zwolnienie przez `free()` przed zakończeniem pracy programu). Niestety w języku C (również C++) nie zaimplementowano mechanizmu odśmiecania pamięci (garbage collection) znanego np. z języka Java. Co więcej, przecieki pamięci, o ile są niewielkie, mogą nie objawiać się w żadem widocznym sposobie i ujawniają się dopiero gdy aplikacja się rozrośnie (często w dość dramatycznych okolicznościach).

W związku z powyższym na programiste spoczywa obowiązek zapewnienia, że tworzony kod nie generuje przecieków pamięci. Najprostszym sposobem weryfikacji czy uruchamiany kod nie tworzy przecieków jest, w przypadku korzystania z kompilatora `+gcc+` lub `clang`, wykorzystanie jego opcji `-fsanitize=leak` (w starszych wersjach tych kompilatorów opcja nie jest dostępna) np.:

```
$ gcc -Wall -fsanitize=leak test.c -o test
```

W środowisku UNIX/Linux można również skorzystać do wykrywania przecieków z narzędziem `mtrace` lub `valgrind`.

W przypadku Windows'owego kompilatora MSVC (Microsoft Visual C++) wyłapywanie przecieków możliwe jest za pomocą dodania wywołania funkcji `_CrtDumpMemoryLeaks()` w ostatniej linii swojego kodu (ważne aby również koniecznie dołączyć plik nagłówkowy `<crtdbg.h>`, np.:

```
#define _CRTDBG_MAP_ALLOC // ustawia wywolania malloc() i free() na debug, co ułatwia wykrywania
// przecieków
#include <stdlib.h>
#include <crtdbg.h>
#include <stdio.h>

int main() {
    // wlaczenie wykrywania przeciekow (ustawienie flag)
    _CrtSetDbgFlag(_CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF);

    // celowo generujemy przeciek (malloc() bez poznieszego free())
    int *przeciek = (int*)malloc(sizeof(int) * 10);

    printf("Test przecieku pamieci\n");

    // Raportujemy przecieki (pojawia sie w oknie wynikowym Visuala (Output))
    _CrtDumpMemoryLeaks();

    return 0; // w tym momencie przeciek zostanie zaraportowany
}
```

W powyższym przykładzie wywołania `#define _CRTDBG_MAP_ALLOC` oraz `_CrtSetDbgFlag()` są opcjonalne.

1.5 Struktury

Obiekt składający się z kilku zmiennych różnych typów nazywany jest w języku C strukturą.

1.5.1 Deklaracja struktury i definicja zmiennych strukturalnych

Deklaracja struktury składa się ze słowa kluczowego `struct`, opcjonalnie występującego po nim identyfikatora (nazwy struktury) oraz z ujętej w nawiasy klamrowe listy składowych oddzielonych średnikami. Deklaracja struktury musi być zakończona średnikiem.

```
struct [struct_name]
{
    type_1 field_1;
    type_2 field_2;
    ...
    type_n field_n;
};
```

Składowe struktury mogą być dowolnego typu prostego lub złożonego, a w szczególności tego samego co typ deklarowanej struktury. W takim przypadku mamy do czynienia ze strukturą rekursywną. O takich konstrukcjach będzie mowa w kolejnej części opracowania. Deklaracja struktury definiuje typ. Zmienne tego typu można definiować w dwojakim sposobie. Pierwszy polega na umieszczeniu nazwy zmiennej (listy nazw) przed średnikiem kończącym deklarację struktury.

```
struct
{
    type_1 field_1;
    type_2 field_2;
    ...
    type_n field_n;
} var_1, var_2, ..., var_n;
```

Jeżeli deklaracja struktury zawiera nazwę, to może być ona użyta do definiowania zmiennych w zwyczajny sposób.

```
struct struct_name var_1, var_2, ... var_n;
```

Sama deklaracja struktury nie rezerwuje pamięci, a tylko opisuje wzorzec typu złożonego. Zmienne strukturalne można inicjalizować umieszczając po ich definicji listę wartości początkowych będących stałymi.

```
struct struct_name var_1, var_2, ..., var_n = {const_1, const_2, ..., const_n};
```

1.5.2 Operacje na strukturach i ich składowych

Do składowej struktury można odwołać się poprzez operator . (kropka). Umożliwia to następująca konstrukcja:

```
struct_name.field
```

Dodatkowo dozwolone jest przypisywanie struktur, ich kopiowanie, pobranie adresu za pomocą operatora &, przekazywanie do funkcji oraz zwracanie jako wartości funkcji. Struktur nie można porównywać. Zwykle, w przypadku konieczności przesłania struktury do funkcji, przekazuje się wskaźnik zamiast kopiować całą jej zawartość. Wskaźnik do struktury definiuje się tak samo jak do innych obiektów.

```
struct struct_name *pp;
```

W takim przypadku pp jest adresem struktury, a (*pp).field jej składową (należy pamiętać o umieszczeniu nawiasów, gdyż operator składowej struktury ma wyższy priorytet od operatora adresowania pośredniego wyłuskania). Bardzo częste stosowanie wskaźników do struktur spowodowało wprowadzenie specjalnego operatora ->. Jeżeli pp jest wskaźnikiem (adresem) do struktury, to pp->field jest jej składową.

1.6 Przykłady konstrukcji rekursywnych

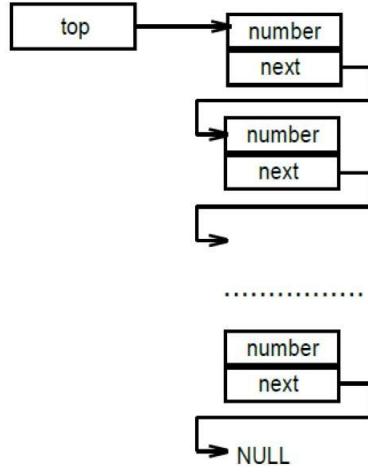
Konstruowanie rekursywnych typów danych umożliwiaowość struktur pozwalająca na deklarowanie składowych o dowolnym typie złożonym. Przeanalizujmy prosty przykład, którym jest stos (kolejka LIFO). Zadeklarujmy strukturę opisującą stos przechowujący obiekty typu `integer`.

```
struct stack_struct
{
    int number ;
    struct stack_struct *next;
} *top = NULL ;
```

Powyższa struktura zawiera dwie składowe. Pierwszą z nich jest przechowywana liczba (pole `number`), drugą adres (wskaźnik) następnego elementu (pole `next`). Należy zwrócić uwagę, że jej typem jest wskaźnik na deklarowaną strukturę. Elementy stosu umieszczone są w dowolnych miejscach przestrzeni adresowej. Wskaźniki pozwalają w każdej sytuacji zlokalizować następny element. Przyjęcie takiego rozwiązania pozwala na połączenie zbioru elementów w jednolitą listę. Wskaźnik `next` ostatniego elementu listy ma wartość `NULL`. Pozwala to na zidentyfikowanie tego elementu przez funkcję przeszukującą listę. Zdefiniowana została również zmienna `top` wskazująca wierzchołek stosu. Jej inicjalną wartością jest `NULL` (oznacza to, że stos jest inicjalnie pusty). Konstruowaną strukturę obrazuje rys.1.1.

Możemy przystąpić do napisania funkcji operujących na opisanej strukturze. Na stosie można wykonać dwie operacje: umieszczenie elementu na jego wierzchołku (`push`) i zdjęcie elementu z wierzchołka (`pop`). Operacje te będą realizowane odpowiednio przez funkcje `push()` oraz `pop()`.

```
void push (int element )
{
    struct stack_struct *p;
    p = top;
    top = (struct stack_struct*) malloc (sizeof (struct stack_struct ));
```



Rys. 1.1: Struktura stosu (dynamicznej listy LIFO)

```

    top->number = element;
    top->next = p;
}

int pop( void )
{
    int element = 0;
    struct stack_struct *p;
    if (top != NULL)
    {
        p = top;
        element = top->number;
        top = top->next;
        free (p);
    }
    return element;
}

```

Funkcja `push()` zapamiętuje w zmiennej tymczasowej aktualną wartość wskaźnika wierzchołka stosu. Następnie alokuje blok pamięci dla nowego elementu i jego adres podstawia do wskaźnika wierzchołka stosu (wskazuje on zawsze na ostatnio wpisany element). Następnie wartość umieszczana na stosie jest wpisywana do składowej `number`. Wskaźnik na następny element jest inicjowany poprzednią wartością wierzchołka stosu.

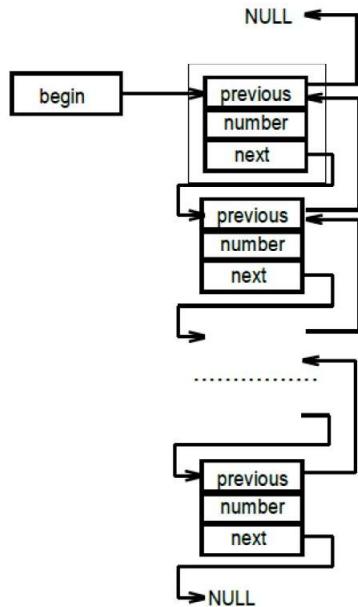
Funkcja `pop()` sprawdza, czy stos nie jest pusty, porównując wskaźnik wierzchołka z wartością `NULL`. Następnie zapamiętuje w zmiennych tymczasowych wartości odczytywanego elementu oraz aktualnego wskaźnika wierzchołka stosu. W kolejnym kroku zostaje przerwane połączenie zdejmowanego elementu ze stosem poprzez podstawienie pod wskaźnik wierzchołka adresu następnego elementu. Na końcu zostaje zwolniony blok pamięci.

Innym, nieco bardziej złożonym, przykładem jest dynamiczna lista liczb całkowitych ułożonych rosnąco. Zadeklarujemy strukturę opisującą element listy dwukierunkowej.

```

struct element
{
    struct element *previous;
    int number;
    struct element *next;
}

```



Rys. 1.2: Struktura listy dwukierunkowej

```
}*begin;
```

Powyższa struktura zawiera trzy składowe. Składowa `number` jest liczbą przechowywaną na danej pozycji listy. Dwie pozostałe są wskaźnikami odpowiednio do poprzedniego i następnego elementu. Podobnie jak w przypadku stosu, elementy listy rozmieszczone są w dowolnych miejscach pamięci operacyjnej. Wskaźniki (zawierające adresy sąsiednich elementów) pozwalają zlokalizować element poprzedzający i następny. Przyjęcie takiego rozwiązania pozwala na połączenie zbioru elementów w listę. W odróżnieniu od stosu, wprowadzone są dwa wskaźniki: na element poprzedni (`previous`) i na element następny (`next`). Takie rozwiązanie pozwoli później na pisanie funkcji przeglądających listę w obie strony.

Konstruowana struktura schematycznie przedstawiona jest na rys.1.2. Pierwszy element listy (zakreślony linią) jest wyróżniony. Jest on umieszczany na liście inicjalnie, gdyż wymaga się aby zawierała ona co najmniej jeden element. Nie przechowuje on danej. Wskaźniki `previous` pierwszego oraz `next` ostatniego elementu listy mają wartość `NULL`. Umożliwia to zidentyfikowanie tych elementów przez funkcję przeszukującą listę. Dodatkowo wprowadzony został wskaźnik `begin` pozwalający zlokalizować pierwszy element listy.

Tak zadeklarowana struktura pozwoli na napisanie funkcji wpisującej liczbę na listę z zachowaniem rosnącego uporządkowania.

```

struct element * insert (int obj , struct element *ptr)
{
    struct element *p;
    p = ptr->next; /* wez adres analizowanego elementu */

    if (p!= NULL) /* czy koniec listy */
    {
        if (obj>(p->number))
        {
            p->next = insert(obj,p);

```

```

    }
    else
    {
        /* utwórz nowy element */
        p->previous = (struct element *) malloc(sizeof(struct element));

        /* zapamiętaj adres nowego elementu */
        p = p->previous;

        /* zainicjuj składowe nowego elementu */
        p->number = obj;
        p->next = ptr->next;
        p->previous = ptr;
    }
}
else /* koniec listy */
{ /* utwórz nowy element */
    p = (struct element *) malloc(sizeof(struct element));

    /* zainicjuj składowe nowego elementu */
    p->number = obj;
    p->next = NULL;
    p->previous = ptr;
}
return p; /* zwroc adres nowego elementu */
}

```

Zdefiniowana wyżej funkcja wstawiająca liczby do dwukierunkowej listy uporządkowanej ma dwa parametry formalne. Pierwszym z nich jest wstawiana liczba, drugim wskaźnik do elementu listy poprzedzającego aktualnie analizowany. Funkcja zwraca adres nowo utworzonego elementu. Skonstruowana jest w sposób rekurencyjny. Jest to praktyka zwykle stosowana przy manipulacji rekursywnymi strukturami danych.

Inicjalnie funkcja wołana jest z parametrem `begin` (wskaźnikiem na początek listy), a wartość zwracana przez funkcję podstawianą jest pod adres drugiego elementu listy (`begin->next`). W pierwszym kroku pod zmiennej pomocniczą `p` podstawiany jest adres elementu następnego (ponieważ analizowany będzie element następny za wskazywanym przez parametr funkcji). Następnie funkcja sprawdza, czy kolejny element istnieje (czy wskaźnik jest różny od `NULL`). Jeśli tak, to wstawiana liczba porównywana jest z przechowywaną w tym elemencie listy. Jeśli wstawiana liczba jest większa, to funkcja jest wołana rekurencyjnie z adresem następnego elementu listy. Jeżeli wstawiana liczba jest mniejsza, bądź równa, to funkcja tworzy nowy element po wskazywanym przez parametr aktualny `ptr`. W tym celu rezerwowany jest odpowiedni obszar pamięci, a jego adres wpisywany do wskaźnika na element poprzedni od analizowanego. Do składowych nowo utworzonego elementu są wpisywane odpowiednie wartości. Jeżeli wskaźnik do kolejnego elementu jest pusty, to osiągnięty został koniec listy (lista była pusta albo wpisywana liczba okazała się większa od wszystkich przechowywanych w liście). W takim przypadku należy utworzyć nowy element, umieścić w nim liczbę, wskaźnik na element następny zainicjować wartością `NULL`, a do wskaźnika na element poprzedni wpisać wartość drugiego parametru funkcji (`ptr`), czyli adres poprzedniego elementu listy.

Uwaga! przed wywołaniem funkcji `insert()` musi zostać utworzony pierwszy (fikcyjny) element listy, a oba jego wskaźniki zainicjowane wartością `NULL`. Adres tego elementu powinien być wpisany do zmiennej `begin`.

```

.....
begin = ( struct element *) malloc (sizeof(struct element));
begin->next = NULL ;
begin->previous = NULL ;
.....
begin->next = insert(i, begin);

```

Jeszcze innym przykładem rekursywnej struktury danych jest drzewo binarne przechowujące w węzłach elementy typu `int`. Opisującą je struktura ma postać:

```
struct tree
{
    int element;
    struct tree *left ;
    struct tree *right ;
};
```

Zakładając, że zmienna `root` wskazuje na wierzchołek tak zdefiniowanego drzewa można napisać rekurencyjną funkcję wypisującą wszystkie elementy umieszczone w węzłach.

```
void scan (struct tree root)
{
    if (root != NULL)
    {
        printf ("\n element = %d", root->element );
        scan (root->left);
        scan (root->right);
    }
}
```