Podstawy programowania 1

Tablice wielowymiarowe

Arkadiusz Chrobot

Katedra Systemów Informatycznych

4 grudnia 2024

Plan

1. Tablice wielowymiarowe
2. Indeksowanie elementów
3. Przekazywanie przez parametry
4. Tablice łańcuchów znaków
5. Operacje na macierzach
6. Gra w życie Conway'a
7. Zakończenie

Tablice wielowymiarowe

Tablice wielowymiarowe, to tablice, które wymagają podania więcej niż jednego indeksu, aby określić położenie elementu w ich wnętrzu. Język C podobnie jak inne współczesne języki programowania umożliwia tworzenie takich tablic. Są one deklarowane w bardzo podobny sposób jak tablice jednowymiarowe. Po pierwszej parze nawiasów kwadratowych należy dopisać kolejne. Musi być ich tyle, ile wymiarów ma mieć deklarowana tablica. W każdej takiej parze nawiasów kwadratowych należy podać liczbę elementów przypadających na poszczególny wymiar. Tablice dwuwymiarowe (tablice tablic) można wyobrazić sobie jako prostokątne. Najczęściej służą one do modelowania macierzy. Tablice trójwymiarowe (tablice tablic tablic) wyobrażamy sobie jako prostopadłościenne. Możemy także stosować tablice o wyższej liczbie wymiarów.

Tablice wielowymiarowe

Inicjacja tablic wielowymiarowych

Tablice wielowymiarowe można utworzyć jako zmienne zainicjowane. W takim przypadku możemy pozostawić pustą pierwszą parę nawiasów kwadratowych, dla pozostałych musimy orkeślić liczbę elementów. Wartość tablicy podajemy jako tablicę innych tablic. Najprostszy przypadek takiej inicjacji, to tablica dwuwymiarowa, gdzie elementami tablicy są tablice jednowymiarowe. Przykład pokazuje taki przypadek.

```
int first_matrix[2][3] = {{1,2,3},{4,5,6}};
int second_matrix[][2] = {{1,2},{3,4}};
```

W pierwszym wierszu zainicjowano tablicę dwuwymiarową o dwóch elementach będących tablicami jednowymiarowymi o trzech elementach typu `int`. W drugim wierszu zainicjowano tablicę dwuwymiarową o dwóch elementach będących tablicami jednowymiarowymi o dwóch elementach typu `int`. Proszę zauważyć, że w drugim przypadku zrezygnowano z podania pierwszej liczby elementów - kompilator sam ją ustali na podstawie wartości inicjującej.

Indeksowanie elementów

Aby uzyskać dostęp do dowolnego elementu tablicy wielowymiarowej należy podać wartości wszystkich jego indeksów. Jest ich tyle, ile wymiarów ma tablica. Każdą z nich podajemy w osobnej parze nawiasów kwadratowych. Wartości indeksów muszą być liczbami naturalnymi, niezależnie od tego, czy sam indeks jest wyrażeniem, pojedynczą zmienną, stałą, czy wprost wartością. Nie mogą one także być większe lub równe liczbie elementów dla danego wymiaru. Przykład zawiera odwołania kolejno do tablicy dwu-wymiarowej, trójwymiarowej i czterowymiarowej, których elementy są typu `double`.

```
double value = matrix[1][0];
value = cube[3][5][7];
value = hypercube[1][3][2][3];
```

Przekazywanie przez parametry

Tablicę wielowymiarową możemy przekazać do funkcji powtarzając jej deklarację na liście parametrów. W miejscu wywołania podstawiamy za ten parametr nazwę tablicy. Możemy również w takiej deklaracji parametru opuścić pierwszą liczbę elementów. Pozostałe musimy podać. Można również nie podawać w parametrze pierwszego wymiaru, a zamiast niego użyć wskaźnika. Na kolejnych trzech slajdach umieszczono przykłady funkcji, które pobierają przez parametry tablice dwuwymiarowe, korzystając z opisanych wyżej sposobów.

Przekazywanie przez parametry

Pierwszy sposób

```
void print_matrix(int matrix[ROWS][COLUMNS])
{
    int i,j;
    for(i=0;i<ROWS;i++) {
        for(j=0;j<COLUMNS;j++)
            printf("%4d",matrix[i][j]);
        puts("");
    }

}
```

Przekazywanie przez parametry

Pierwszy sposób - komentarz

Poprzedni slajd zawiera kod źródłowy funkcji wypisującej na ekran zawartość tablicy dwuwymiarowej o elementach typu `int`. Rozmiary tej tablicy zostały opisane za pomocą stałych `ROWS` i `COLUMNS`, co ułatwia ich zmianę. Wystarczy zmienić ich wartości w miejscu definicji stałej. Proszę zwrócić uwagę, że do uzyskania dostępu do każdego elementu tablicy potrzebne są dwie zagnieżdżone pętle. Proszę również zwrócić uwagę na użycie funkcji `puts()`. W tym przykładzie tablica dwuwymiarowa modeluje macierz, a rolą wspomnianej funkcji jest tylko przeniesienie kursora do następnego wiersza ekranu, tak aby wypisać tam kolejny wiersz macierzy. Dzięki temu wygląda ona podobnie jak w zapisie matematycznym. Na wartość każdego elementu tej tablicy funkcja `printf()` przeznacza, zgodnie z zapisem w ciągu formatującym, cztery miejsca na ekranie.

Przekazywanie przez parametry

Drugi sposób

```
void fill_matrix(int matrix[][COLUMNS])
{
    int i,j;
    srand(time(0));
    for(i=0;i<ROWS;i++)
        for(j=0;j<COLUMNS;j++)
            matrix[i][j] = rand()%10;
}
```

Przekazywanie przez parametry

Drugi sposób - komentarz

Funkcja zaprezentowana na poprzednim slajdzie wypełnia macierz liczbami naturalnymi losowanymi z zakresu od 0 do 9. Tym razem w deklaracji parametru opuszczono pierwszą liczbę elementów. Proszę zwrócić uwagę, że stała `ROWS` nadal jest używana w funkcji, a dokładniej w pierwszej pętli `for`.

Przekazywanie przez parametry

Trzeci sposób

```
void print_matrix(int (*matrix)[COLUMNS])
{
    int i,j;
    for(i=0;i<ROWS;i++) {
        for(j=0;j<COLUMNS;j++)
            printf("%4d",matrix[i][j]);
        puts("");
    }
}
```

Przekazywanie przez parametry

Trzeci sposób - komentarz

W funkcji z poprzedniego slajdu użyto wskaźnika do przekazania tablicy dwuwymiarowej przez parametr. Ta funkcja wypisuje macierz, tak jak pierwsza z zaprezentowanych na wykładzie. Proszę zwrócić uwagę na nawiasy okrągłe użyte w deklaracji parametru. Dzięki nim ten wskaźnik jest wskaźnikiem na tablicę a nie tablicą wskaźników.

Tablice łańcuchów znaków

Specjalnym przypadkiem tablic dwuwymiarowych są tablice łańcuchów znaków. Możemy odwoływać się do poszczególnych liter zapisanych w określonym łańcuch przechowywanym w tej tablicy podając dwa indeksy: indeks łańcucha i indeks znaku w łańcuchu. Możemy także podać położenie tylko i wyłącznie łańcucha podając jeden indeks. Takie tablice tworzy się jak zwykłe tablice dwuwymiarowe, podając jako typ elementu `char`. Można także tworzyć je jako tablice zainicjowane. Przykład zawiera deklarację takiej właśnie tablicy.

```
char cities[][LENGTH] = {"Bruksela","Amsterdam","Antwerpia",
                         "Kraków","Wiedeń","Warszawa"};
```

Stała `LENGTH` określa liczbę elementów, które mogą przechowywać znaki łańcucha. Łańcuchy znaków, jako wartości inicjujące nie muszą być zapisywane w osobnych nawiasach klamrowych, wystarczy, że są ujęte w cudzysłów. Operacje na takiej tablicy zostaną zaprezentowane na przykładzie funkcji wypisującej jej zawartość oraz dwóch funkcji sortujących tę tablicę z użyciem algorytmu sortowania przez wybór.

Tablice łańcuchów znaków

Wypisanie zawartości tablicy na ekran

```
void print_cities(char cities[][LENGTH])
{
    int i;
    for(i=0; i<NUMBER; i++)
        printf("%s ",cities[i]);
    puts("");
}
```

Tablice łańcuchów znaków

Wypisanie zawartości tablicy na ekran - komentarz

Zaprezentowana funkcja wypisuje wszystkie łańcuchy z tablicy na ekranie. Tablica przekazywana jest przez parametr, w którym pominięto określenie pierwszej liczby elementów. Liczbę łańcuchów znaków w tablicy określa stała `NUMBER`, użyta w pętli `for`. Do wypisania takiej tablicy na ekran wystarcza jedna instrukcja iteracyjna. Poszczególne ciągi znaków wypisywane są całościowo przez funkcję `printf()` dzięki użyciu ciągu formatującego `"%s"`.

Tablice łańcuchów znaków

Funkcja swap()

```
void swap(char first[], char second[])
{
    char tmp[LENGTH];

    strncpy(tmp,first,LENGTH-1);
    strncpy(first,second,LENGTH-1);
    strncpy(second,tmp,LENGTH-1);
}
```

Tablice łańcuchów znaków

Funkcja `swap()` - komentarz

Wersja funkcji `swap()` dla tablicy łańcuchów znaków jest inna niż dla tablicy prostych typów, takich jak np. `int`. Jako parametry używane są tablice znaków. W miejscu wywołania tej funkcji będą pod nie podstawione elementy tablicy łańcuchów znaków. Wartości tych elementów zostaną przez funkcję zamienione miejscami. W tym celu tworzona jest zmienna `tmp`[1], będąca tablicą do przechowywania łańcuchów znaków. Kopiowanie łańcuchów zrealizowane jest za pomocą funkcji `strncpy()`. Stała `LENGTH` została użyta w wyrażeniu określającym maksymalną liczbę znaków, jakie ta funkcja może skopiować.

[1] Ta nazwa jest skrótem od angielskiego słowa *temporary*, które oznacza po polsku „tymczasowo".

Tablice łańcuchów znaków

Sortowanie tablic ciągów znaków

```
void sort_cities(char (*cities)[LENGTH])
{
    int i,j;

    for(i=0; i<NUMBER-1; i++) {
        int min = i;
        for(j=i+1; j<NUMBER; j++)
            if(strncmp(cities[min],cities[j],LENGTH-1)>0)
                min = j;
        if(min!=i)
            swap(cities[min],cities[i]);
    }
}
```

Tablice łańcuchów znaków

Sortowanie tablic ciągów znaków - komentarz

Funkcja z poprzedniego slajdu jest przerobioną wersją funkcji sortującej tablice jednowymiarowe z użyciem algorytmu sortowania przez wybór. W nagłówku, poza nazwą funkcji został zmieniony jej parametr, aby można było za jego pomocą przekazać do funkcji dwuwymiarową tablice znaków, czyli tablicę łańcuchów znaków. Zmianie ulega nie tylko nazwa tego parametru, ale przede wszystkim jego typ. Zmieniono także identyfikator stałej określającej liczbę elementów tablicy. Ważniejszą zmianą jest jednak zastosowanie w instrukcji warunkowej funkcji `strncmp()` celem określenia, czy wartość elementu wskazywanego przez indeks `min` nie jest większa od tego, który wskazuje indeks `j`. Jeśli tak jest, to funkcja ta zwraca wartość większą od zera.

Tablice łańcuchów

Argumenty wywołania programu

Tablice łańcuchów mają szczególne zastosowanie w przypadku funkcji `main()`. Otóż jej lista parametrów nie musi być pusta. Może ona zawierać dwa parametry, które zwyczajowo (nazwy te można zmienić) nazywają się `argc` i `argv`. Pierwszy jest typu `int`, drugi jest tablicą wskaźników na łańcuchy znaków. Oba służą do przekazywania do funkcji `main()` *argumentów wywołania programu*. Przykładem takich argumentów są opcje poleceń wywoływanych za pomocą powłoki systemowej w systemach Unix i pokrewnych. Przez program są one traktowane jako łańcuch znaków. Jeśli jest ich kilka, to są rozdzielone znakami białymi, np. spacjami. Te łańcuchy zostają zapisane w tablicy `argv`, a ich liczba w parametrze `argc`. Należy pamiętać, że ten ostatni parametr zawsze ma wartość co najmniej równą jeden, gdyż zawsze pierwszy element `argv` zawiera pełną nazwę wywołanego programu, tzn. nazwę jego pliku wykonywalnego wraz ze ścieżką dostępu. Na następnym slajdzie znajduje się kod programu, który wypisuje na ekran w kolejnych wierszach wartości argumentów z jakimi został wywołany.

Tablice łańcuchów

Argumenty wywołania programu

```c
#include<stdio.h>

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

Operacje na macierzach

Tablice dwuwymiarowe używane są do reprezentowania macierzy. W takim przypadku pierwsza liczba elementów podawana w deklaracji macierzy określa liczbę jej wierszy, a druga liczbę kolumn i tym samym liczbę elementów w każdym wierszu. Zatem macierz o wymiarach 2×3 i elementach będących liczbami całkowitymi może być zadeklarowana następująco:

```
int matrix[2][3];
```

Jeśli obie liczby elementów są sobie równe, to macierz jest macierzą kwadratową. W odwołaniu do elementu pierwszy indeks określa wiersz, drugi indeks określa kolumnę, w których położony jest element. Operacje jakie można wykonywać na macierzy to m.in.: dodawanie, odejmowanie, mnożenie, transponowanie, odwracanie i wyznaczanie wyznacznika. Na kolejnych slajdach zostaną opisane funkcje, które realizują pierwsze cztery.

Dodawanie

Dodawanie możliwe jest tylko w przypadku macierzy o takich samych wymiarach. Macierz wynikowa ma te same wymiary co argumenty. Dodawanie macierzy polega na dodaniu do siebie wartości odpowiadających sobie elementów obu argumentów. Ilustruje to następny slajd. Zaznaczono na nim trzy elementy macierzy. Na zielono i niebiesko zaznaczone są dodawane elementy, a na czerwono element wynikowy.

Operacje na macierzach

Dodawanie

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} + \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} = \begin{bmatrix} 2 & 4 & 6 \\ 8 & 10 & 12 \end{bmatrix}$$

Operacje na macierzach

Dodawanie - implementacja

```
void add_matrices(int (*argument_1)[COLUMNS],
                  int (*argument_2)[COLUMNS],
                  int result[ROWS][COLUMNS])
{
    int i,j;
    for(i=0;i<ROWS;i++)
        for(j=0;j<COLUMNS;j++)
            result[i][j] =
                argument_1[i][j] + argument_2[i][j];
}
```

Operacje na macierzach

Dodawanie - komentarz do implementacji

Macierze nie mogą być, przynajmniej bezpośrednio, zwracane jako wynik działania funkcji. Zatem funkcja `add_matrices` posiada trzy parametry. Przez pierwsze dwa przekazywane są do niej macierze, które należy dodać, przez trzeci zwracana jest macierz wynikowa. Dodawanie wymaga dwóch zagnieżdżonych pętli. Pierwsza indeksuje wiersze we wszystkich trzech macierzach, a druga konkretne elementy należące do tego wiersza.

Operacje na macierzach

Odejmowanie

Odejmowanie macierzy wykonywane jest analogicznie do dodawania. Kolejny slajd pokazuje przykład takiej operacji.

Operacje na macierzach

Odejmowanie

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} - \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Operacje na macierzach

Odejmowanie - implementacja

```
void subtract_matrices(int (*argument_1)[COLUMNS],
                       int (*argument_2)[COLUMNS],
                       int result[ROWS][COLUMNS])
{
    int i,j;
    for(i=0;i<ROWS;i++)
        for(j=0;j<COLUMNS;j++)
            result[i][j] =
                argument_1[i][j] - argument_2[i][j];
}
```

Odejmowanie - komentarz do implementacji

Funkcja implementująca odejmowanie jest napisana analogicznie do funkcji implementującej dodawanie. Jedyne różnice, to nazwa funkcji i użyty operator (odejmowanie zamiast dodawania).

Operacje na macierzach

Mnożenie

Mnożenie macierzy jest bardziej skomplikowaną operacją niż dodawanie i odejmowanie. Różni je od nich to, że nie jest ono przemienne. Macierz będąca pierwszym argumentem mnożenia musi mieć tyle samo kolumn, ile wierszy ma macierz będąca drugim argumentem. Wyznaczanie wartości pojedynczego elementu macierzy wynikowej polega na zsumowaniu iloczynów wartości wszystkich elementów określonego wiersza pierwszej macierzy i określonej kolumny drugiej macierzy. Ilustruje to przykład na następnym slajdzie. Wartość elementu zaznaczonego na czerwono uzyskano mnożąc zaznaczony na zielono wiersz z pierwszej macierzy, przez zaznaczoną na niebiesko kolumnę z drugiej macierzy, czyli wykonano działanie $1 * 1 + 2 * 3 + 3 * 5 = 22$.

Operacje na macierzach

Mnożenie

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} = \begin{bmatrix} 22 & 28 \\ 49 & 64 \end{bmatrix}$$

Operacje na macierzach

Mnożenie - implementacja

```
void multiply_matrices(int argument_1[2][3],
                       int argument_2[3][2],
                       int result[2][2])
{
    int i,j,k;
    for(i=0;i<2;i++)
        for(j=0;j<2;j++)
            for(k=0;k<3;k++)
                result[i][j] +=
                    argument_1[i][k]*argument_2[k][j];
}
```

Mnożenie - komentarz do implementacji

Parametry funkcji mają takie samo znaczenie, jak w przypadku poprzednich funkcji implementujących operacje na macierzach. Do mnożenia wykorzystany został najprostszy możliwy algorytm. Wymaga on użycia trzech pętli. Pierwsza wskazuje wiersz w pierwszym argumencie, druga wskazuje kolumnę w drugim argumencie, a trzecia indeksuje kolejne elementy tego wiersza i tej kolumny. Oprócz tego liczniki dwóch pierwszych pętli określają element w macierzy wynikowej, do którego powinien trafić wynik działania. Ponieważ jest on również używany do sumowania iloczynów poszczególnych elementów, to jego wartość początkowa powinna wynosić zero. Zakładamy zatem, że trzeci argument wywołania funkcji będzie macierzą, której wszystkie elementy będą wyzerowane. Wystarczy, aby ta macierz była zmienną globalną.

Operacje na macierzach

Efektywność mnożenia macierzy

Kolejność użytych w funkcji pętli można dowolnie zmieniać. Okazuje się, że ma to wpływ na czas działania tej funkcji. Jeśli będziemy identyfikować te pętle po nazwach ich zmiennych sterujący, to najczęściej najkorzystniejszą konfiguracją jest (*ikj*), choć zależy to także od konfiguracji komputera, na którym ta funkcja jest uruchamiana. Istnieją inne, bardziej efektywne algorytmy mnożenia macierzy, ale wzrost efektywności jest zauważalny dopiero dla bardzo dużych macierzy. Dodatkowo te algorytmy są trudne do prawidłowego zaimplementowania.

Operacje na macierzach

Transponowanie

Transponowanie macierzy w najprostszym ujęciu polega na zamianie miejscami wierszy z kolumnami, czyli np. macierz $\mathbb{A}$ o wymiarach 2×3 staje się macierzą o wymiarach 3×2. Operację transponowania macierzy oznaczamy w matematyce następująco: $\mathbb{A}^T$. Następny slajd zawiera przykład takiej operacji.

Operacje na macierzach

Transponowanie

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}^T = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

Operacje na macierzach

Transponowanie - implementacja

```
void transpose_matrix(int argument[ROWS][COLUMNS],
                      int result[COLUMNS][ROWS])
{
    int i,j;
    for(i=0;i<2;i++)
        for(j=0;j<3;j++)
            result[j][i]=argument[i][j];
}
```

Transponowanie - komentarz do implementacji

Transponowanie macierzy prostokątnej wymaga użycia drugiej macierzy do zapisania wyniku. Macierz, która ma być transponowana, przekazywana jest przez pierwszy parametr funkcji, a wynik zwracany jest za pomocą drugiego parametru. Proszę zwrócić uwagę na sposób zapisu indeksów w macierzy wynikowej. Ich kolejność jest odwrotna niż w przypadku macierzy będącej argumentem.

Operacje na macierzach

Transponowanie macierzy kwadratowej - implementacja

```
void transpose_square_matrix(int matrix [3][3])
{
    int i,j;
    for(i=0; i<3; i++)
        for(j=0; j<i; j++) {
                int tmp;
                tmp = matrix[i][j];
                matrix[i][j] = matrix[j][i];
                matrix[j][i] = tmp;
        }
}
```

Operacje na macierzach

Transponowanie macierzy kwadratowej - komentarz do implementacji

W przypadku transponowania macierzy kwadratowej operację tę można przeprowadzić używając tylko jednej macierzy. Wystarczy zamienić miejscami wartości elementów macierzy trójkątnych wyznaczanych przez przekątną. Proszę zwrócić uwagę na konstrukcję wewnętrznej pętli. Zakres wartości jej licznika (zmienna `j`) jest ograniczony przez wartość licznika pętli zewnętrznej (zmienna `i`) w bieżącej iteracji. Operacje wykonywane wewnątrz pętli wewnętrznej, to zamiana wartości dwóch elementów macierzy. Można do tego użyć tej samej funkcji `swap()`, która zastała zdefiniowana na wykładzie poświęconym sortowaniu tablic.

Gra w życie Conway'a

Gra w życie (ang. *game of life*) jest automatem komórkowym (ang. *cellular automaton* w skrócie CA), który został opracowany przez brytyjskiego matematyka Johna Conway'a. Wbrew nazwie nie służy on do rozrywki, przynajmniej nie w tradycyjnym znaczeniu tego słowa. Jest on zbudowany z komórek ułożonych na nieskończonej kwadratowej planszy, które zmieniają swój stan w kolejnych krokach pracy automatu, zgodnie z kilkoma prostymi regułami. Stany wszystkich komórek tworzą stan całej gry, który potrafi ewoluować w bardzo skomplikowany sposób. Automat ten jest przedmiotem badań zarówno matematyków, jak i fizyków, informatyków oraz biologów. Z punktu widzenia informatyki, automat ten jest równoważny maszynie Turinga, a więc jest także komputerem, który przetwarza informacje. Nas interesować będzie inny aspekt tej gry - do jej symulacji możemy wykorzystać macierze.

Gra w życie Conway'a

Reguły ewolucji stanu komórki

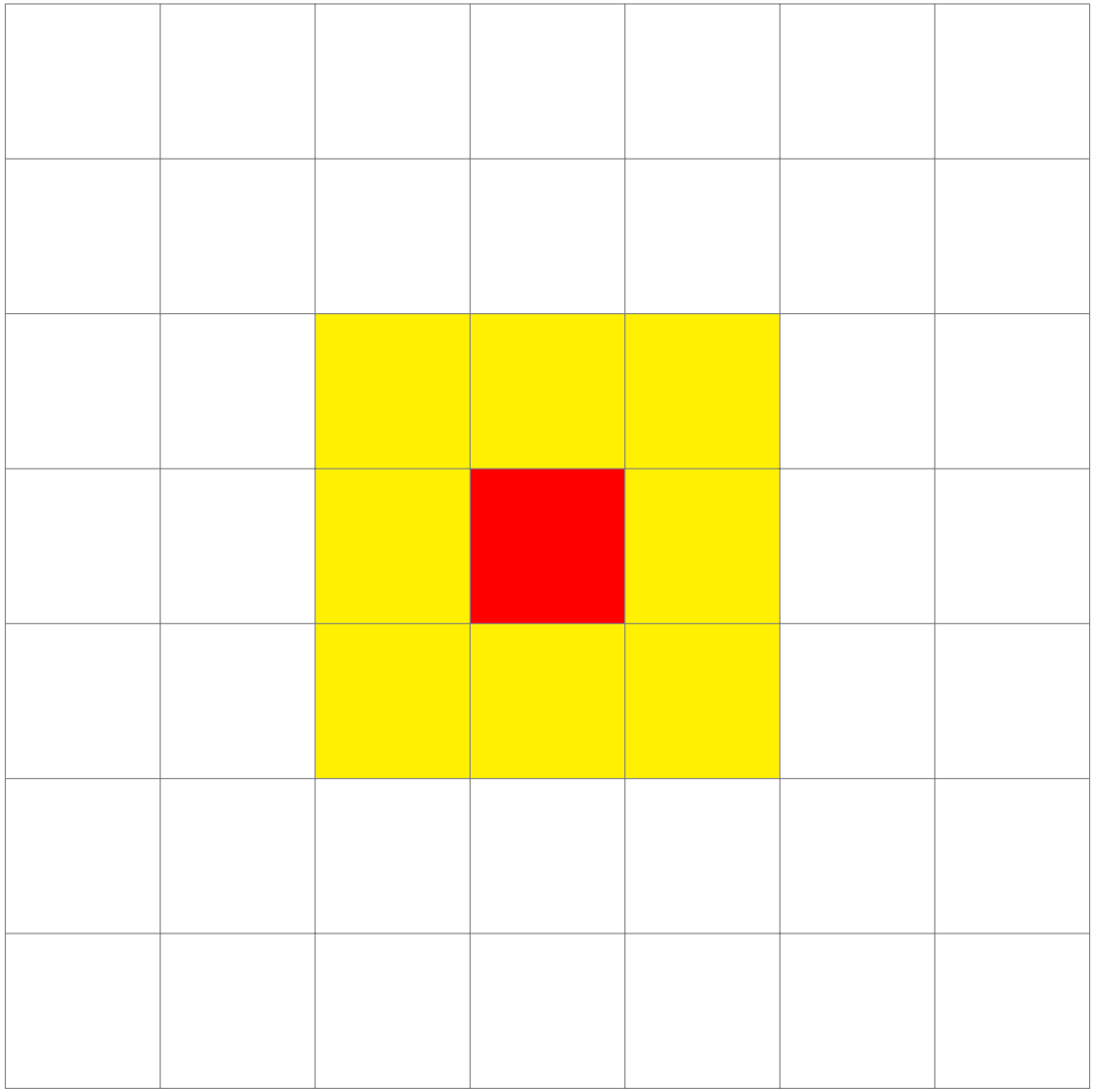
Komórki automatu reprezentowane są za pomocą elementów macierzy. Każda komórka automatu może być w danym kroku w jednym z dwóch stanów: martwa lub żywa. Zmiana stanu komórki w następnym kroku zależna jest od jej stanu i stanów jej sąsiadów w kroku bieżącym oraz podlega następującym regułom:

1. każda żywa komórka, która ma mniej niż dwóch żywych sąsiadów umiera,
2. każda żywa komórka, która ma więcej niż trzech sąsiadów umiera,
3. każda żywa komórka, która ma dwóch lub trzech żywych sąsiadów przeżywa,
4. każda martwa komórka, która ma trzech żywych sąsiadów ożywa.

Następny slajd ilustruje definicję sąsiedztwa jaka jest przyjęta w grze w życie.

Gra w życie Conway'a

Ośmiosąsiedztwo



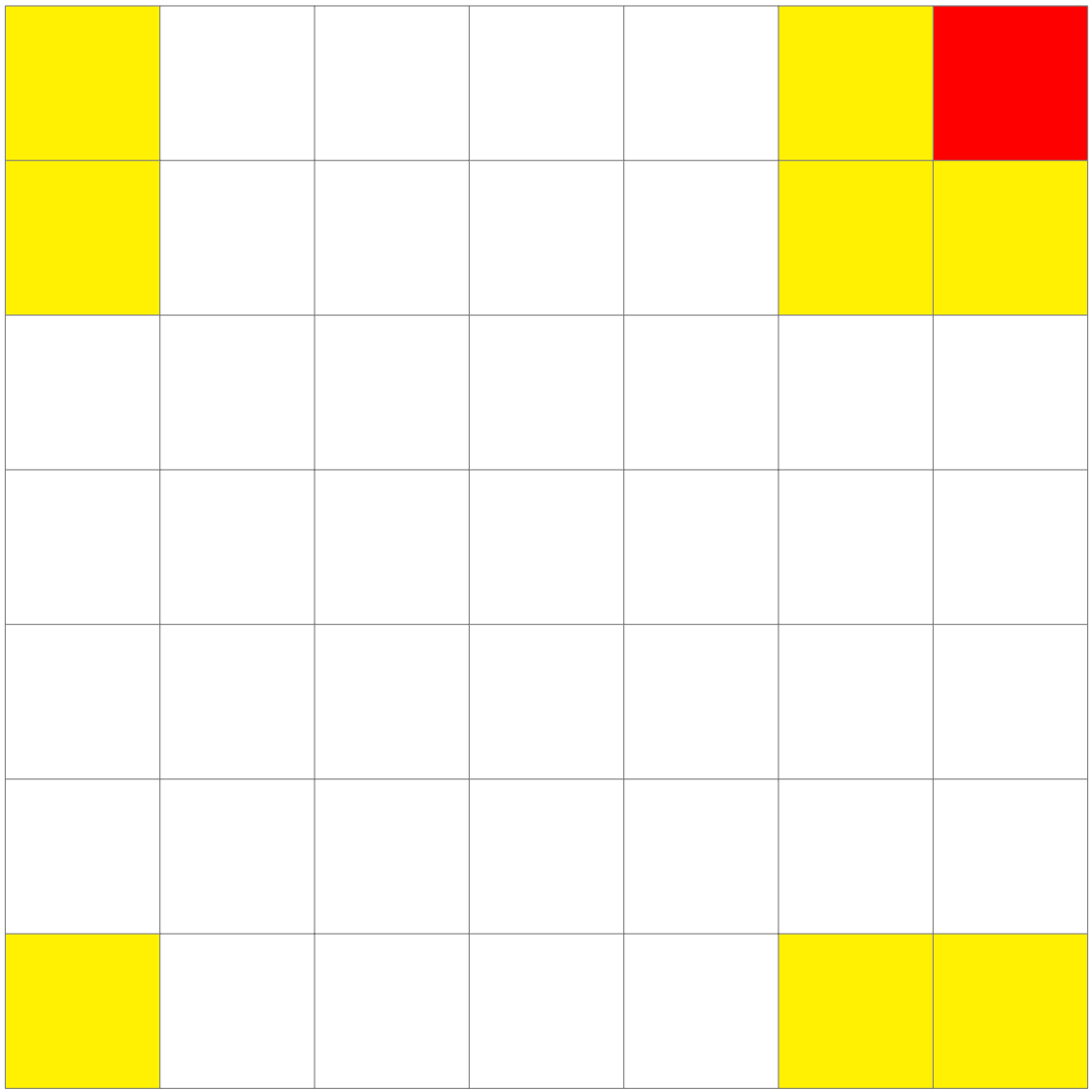
Gra w życie Conway'a

Współrzędne sąsiadów

Jak można się przekonać komórka zaznaczona na czerwono, ma ośmiu sąsiadów, którzy zostali oznaczeni kolorem żółtym. Wyznaczenie ich położenia nie jest trudne. Ich współrzędne (wiersz i kolumna) różnią się co najwyżej o $+1$ lub -1 od współrzędnych komórki wyjściowej. Przykładowo, jeśli komórka wyjściowa ma współrzędne (x, y), to jej górny lewy sąsiad ma współrzędne $(x-1, y-1)$, przy założeniu, że punkt $(0, 0)$ znajduje się w lewym górnym rogu macierzy, indeksy wierszy rosną „w dół", a indeksy kolumn w prawo. W tym rozumowaniu nie uwzględniliśmy jednak pewnego problemu, który ilustruje następny slajd.

Gra w życie Conway'a

Ośmiosąsiedztwo



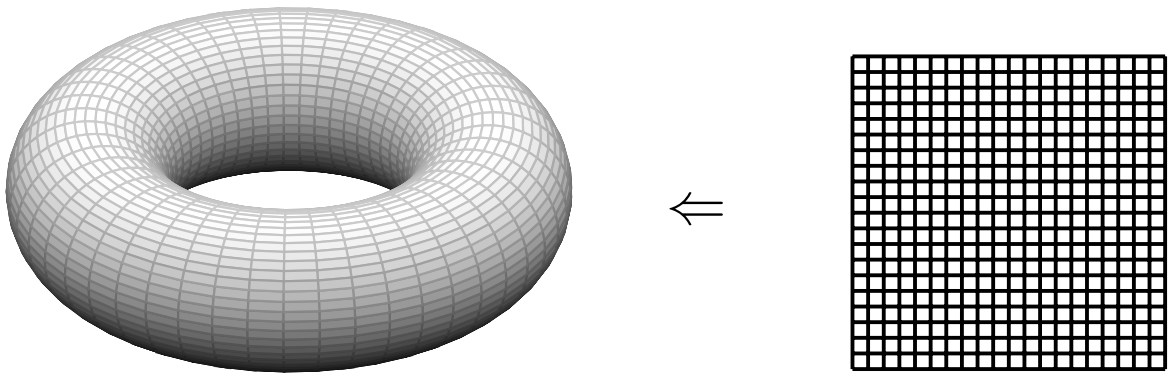
Gra w życie Conway'a

Współrzędne sąsiadów

Nie uwzględniliśmy, że plansza ma być nieskończona. Oznacza to, że elementy znajdujące się na „brzegach" macierzy, też muszą mieć swoich sąsiadów i to dokładnie ośmiu. Można tę trudność pokonać stosując arytmetykę modularną, co zostanie zaprezentowane w kodzie źródłowym programu. Będziemy zatem dążyć do tego, aby tę macierz kwadratową przekształcić w torus, co obrazuje następny slajd.

Gra w życie Conway'a

Plansza



Gra w życie Conway'a

Implementacja

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

#define SIZE 32

enum state {DEAD, ALIVE};

unsigned char board[SIZE][SIZE];
```

Gra w życie Conway'a

Implementacja - komentarz

Zamieszczony na poprzednim slajdzie fragment kodu źródłowego programu oprócz instrukcji włączających odpowiednie pliki nagłówkowe zawiera także definicję stałej określającej liczbę elementów każdego z wymiarów macierzy (32 elementy), definicję typu wyliczeniowego oraz deklarację macierzy, która będzie modelowała planszę, na której rozgrywać się będzie gra. Nie będziemy używać w programie zmiennych typu wyliczeniowego. Został on wprowadzony do programu po to, aby nadać nazwy dwóm możliwym stanom komórki. Komórka martwa ma stan o wartości zero (`DEAD`), a żywa ma stan równy jeden (`ALIVE`).

Gra w życie Conway'a

Implementacja

```
unsigned int down(unsigned int y)
{
    return (y+1)%SIZE;
}

unsigned int up(unsigned int y)
{
    return (y+(SIZE-1))%SIZE;
}
```

Gra w życie Conway'a

Implementacja - komentarz

Funkcje `up()` i `down()` pozwalają określić składowe pionowe współrzędnych sąsiadów dowolnej komórki na planszy. W funkcji `down()` zastosowano operator reszty z dzielenia, aby dla komórek położonych „maksymalnie u dołu" macierzy wyznaczyć sąsiadów znajdujących się „maksymalnie u góry". Odwrotnie działa funkcja `up()`. Funkcja `down()` zwiększa wartość składowej pionowej współrzędnych komórki o jeden w zakresie od `0` do `SIZE-1`, a funkcja `up()` zmniejsza tę wartość o jeden w tym samym zakresie.

Gra w życie Conway'a

Implementacja

```
unsigned int right(unsigned int x)
{
    return (x+1)%SIZE;
}

unsigned int left(unsigned int x)
{
    return (x+(SIZE-1))%SIZE;
}
```

Gra w życie Conway'a

Implementacja - komentarz

Składowe poziome współrzędnych sąsiadów komórki można by wyznaczać za pomocą tych samych funkcji, które służą do określania współrzędnych pionowych, jednak dla czytelności zostały zdefiniowane dwie kolejne funkcje, które działają analogiczne, ale są używane tylko do wyznaczania składowej poziomej.

Gra w życie Conway'a

Implementacja

```
unsigned char count_alive_neighbours(
                            unsigned char board[SIZE][SIZE],
                            unsigned int i, unsigned int j)
{
    return board[i][down(j)]
           + board[i][up(j)]
           + board[left(i)][j]
           + board[right(i)][j]
           + board[right(i)][down(j)]
           + board[right(i)][up(j)]
           + board[left(i)][down(j)]
           + board[left(i)][up(j)];
}
```

Gra w życie Conway'a

Implementacja - komentarz

Funkcja `count_alive_neighbours()` zlicza ile komórka o współrzędnych `i` i `j` ma żywych sąsiadów. Skoro każda żywa komórka ma wartość `1`, a martwa `0`, to aby dowiedzieć się ile dana komórka ma żywych sąsiadów wystarczy zsumować ich wartości.

Gra w życie Conway'a

Implementacja

```
void get_next_step(unsigned char board[SIZE][SIZE])
{
    static unsigned char swap[SIZE][SIZE];
    unsigned int i,j;

    for(i=0; i<SIZE; i++)
        for(j=0; j<SIZE; j++) {
            unsigned char state = board[i][j];
            unsigned char alive_neighbours = count_alive_neighbours(board,i,j);
            if(state == ALIVE && alive_neighbours < 2)
                swap[i][j] = DEAD;
            if(state == ALIVE && alive_neighbours > 3)
                swap[i][j] = DEAD;
            if(state == ALIVE && (alive_neighbours == 3 || alive_neighbours == 2))
                swap[i][j] = ALIVE;
            if(state == DEAD && alive_neighbours == 3)
                swap[i][j] = ALIVE;
        }
    memcpy(board,swap,SIZE*SIZE);
}
```

Gra w życie Conway'a

Implementacja - komentarz

Funkcja `get_next_step()` wyznacza stan automatu w kolejnym kroku. Nowy stan zapisywany jest w macierzy `swap` i tworzony na podstawie bieżącego, przekazanego do funkcji w postaci macierzy `board`. Proszę zwrócić uwagę, że macierz `swap` jest zmienną statyczną, aby zapewnić, że na początku gry wszystkie jej elementy będą miały wartość zero. W dwóch pętlach odczytywany jest bieżący stan każdej komórki, wyznaczana jest liczba jej żywych sąsiadów, a następnie, zgodnie z opisanymi wcześniej regułami zapisywany jest w macierzy `swap` stan jaki będzie miała ta komórka w następnym kroku. Po zakończeniu wykonania pętli zawartość macierzy `swap` kopiowana jest do macierzy `board` za pomocą funkcji `memcpy()`.

Gra w życie Conway'a

Implementacja

```
void seed_board(unsigned char board[SIZE][SIZE])
{
    unsigned int i,j,k;
    srand(time(NULL));
    for(k = 0; k<8; k++) {
        i = rand()%SIZE;
        j = rand()%SIZE;
        board[i][j] = ALIVE;
        int choice = rand()%8;
        switch(choice) {
        case 0 :
            board[i][down(j)] = ALIVE;
        case 1 :
            board[i][up(j)] = ALIVE;
        case 2 :
            board[left(i)][j] = ALIVE;
        case 3 :
            board[right(i)][j] = ALIVE;
        case 4 :
            board[right(i)][down(j)] = ALIVE;
        case 5 :
            board[right(i)][up(j)] = ALIVE;
        case 6 :
            board[left(i)][down(j)] = ALIVE;
        case 7 :
            board[left(i)][up(j)] = ALIVE;
        }
    }
}
```

Gra w życie Conway'a

Implementacja - komentarz

Funkcja `seed_board()` inicjuje planszę przed początkiem gry. Losowane są w niej współrzędne ośmiu komórek, którym nadawany jest stan `ALIVE` oraz liczba sąsiadów dla każdej z tych komórek, którym nadawany jest status żywych.

Gra w życie Conway'a

Implementacja

```
void create_blinker(unsigned char board[SIZE][SIZE])
{
    board[SIZE/2-1][SIZE/2-1] = board[SIZE/2][SIZE/2-1]
    = board[SIZE/2+1][SIZE/2-1] = ALIVE;
}
```

Gra w życie Conway'a

Implementacja - komentarz

Funkcja `create_blinker()` również służy do określenia stanu początkowego gry, ale tym razem w sposób deterministyczny. Tworzy ona w środku planszy prosty „organizm" należący do grupy oscylatorów, czyli grupy komórek, które cyklicznie zmieniają swój stan. Blinker, nazywany też „światłami ulicznymi", to trzy komórki. Dwie poziome i dwie pionowe na przemian stają się martwe i żywe. Dzięki temu obserwujemy wzór trzech „krążących" żywych komórek.

Gra w życie Conway'a

Implementacja

```
void create_ten_in_row(unsigned char board[SIZE][SIZE])
{
    memset(&board[SIZE/2-1][SIZE/2-6],ALIVE,10);
}
```

Gra w życie Conway'a

Implementacja - komentarz

Funkcja `create_ten_in_row()` również inicjuje planszę umieszczając na jej środku „organizm" składający się z dziesięciu żywych komórek umieszczonych obok siebie w poziomie. Jego nazwa pochodzi od tego układu, choć w niektórych opracowaniach nazywany jest również „krokodylem". Okazuje się, że ten prosty organizm ewoluuje w dosyć skomplikowany sposób. Do nadania wartości `ALIVE` dziesięciu komórkom naraz wykorzystano funkcję `memset()`. Umożliwia to budowa macierzy - każda jej komórka ma rozmiar jednego bajta, zatem `memset()` po prostu nadaje wartość przekazaną jej jako drugi argument wywołania dziesięciu kolejnym elementom macierzy, począwszy od elementu o współrzędnych $(\texttt{SIZE}/2 - 1, \texttt{SIZE}/2 - 6)$.

Gra w życie Conway'a

Implementacja

```
void print_board(unsigned char board[SIZE][SIZE])
{
    unsigned int i,j;

    for(i=0; i<SIZE; i++) {
        for(j=0; j<SIZE; j++)
            printf("%2d",board[i][j]);
        printf("\n");
    }
}
```

Gra w życie Conway'a

Implementacja - komentarz

Funkcja `print_board()` służy do wypisania bieżącego stanu gry, czyli zawartości macierzy przekazanej jej przez parametr na ekran. Na każdą wartość komórki są rezerwowane dwa miejsca na ekranie. Proszę zwrócić uwagę na użycie funkcji `printf()` zamiast `puts()` do przenoszenia kursora do następnego wiersza na ekranie.

Gra w życie Conway'a

Implementacja

```
int main(int argc, char **argv)
{
    if(argc==2) {
        if(!strcmp(argv[1],"blinker"))
            create_blinker(board);
        else if(!strcmp(argv[1],"ten_in_row"))
            create_ten_in_row(board);
        else
            seed_board(board);
    } else
        seed_board(board);

    do {
        print_board(board);
        get_next_step(board);
    } while(getchar()!='q');
    return 0;
}
```

Gra w życie Conway'a

Implementacja - komentarz

Funkcja `main()` programu na początku wykonuje inicjację stanu gry na podstawie tego, jakie i czy zostały jej przekazane argumenty wywołania programu. Jeśli został jej przekazany jeden argument, to sprawdza, czy jest on równy łańcuchowi `blinker`, czy `ten_in_row`. W zależności od tego, który z przypadków wystąpił, to taki „organizm" jest tworzony na planszy. Jeśli program został uruchomiony bez argumentów wywołania, lub z nieprawidłowymi argumentami, to plansza jest inicjowana (pseudo)losowo. Po inicjacji wykonywana jest pętla `do...while`. Jest ona wykonywana tak długo, jak długo użytkownik nie naciśnie klawiszy `q` i `Enter`. W tej pętli wyświetlany jest bieżący stan gry za pomocą wywołania funkcji `print_board()`, a następnie generowany jest następny przy użyciu `get_next_step()`.

Gra w życie Conway'a

Implementacja - komentarz

Zaprezentowany program wyświetla stan gry za pomocą serii zer i jedynek, dlatego nie jest być może zbyt „widowiskowy", jednakże można uzyskać efekt płynności po naciśnięciu i przytrzymaniu klawisza `Enter`. Gra w życie, w zależności od stanu początkowego może zakończyć się na cztery sposoby:

1. plansza się wyzeruje - wszystkie komórki staną się martwe,
2. automat osiągnie stan stabilny - na planszy pozostaną organizmy, których stan nie będzie ulegał zmianie,
3. stan automatu będzie ciągle się zmieniał w sposób cykliczny,
4. gra będzie nieskończenie ewoluowała.

W trakcie gry na planszy może powstać szereg „organizmów" (struktur) o ciekawych właściwościach.

Podziękowania

Składam podziękowania dla dra inż. Grzegorza Łukawskiego i dra inż. Leszka Ciopińskiego za udostępnienie materiałów, których fragmenty zostały wykorzystane w tym wykładzie.

Pytania

?

KONIEC

Dziękuję Państwu za uwagę.