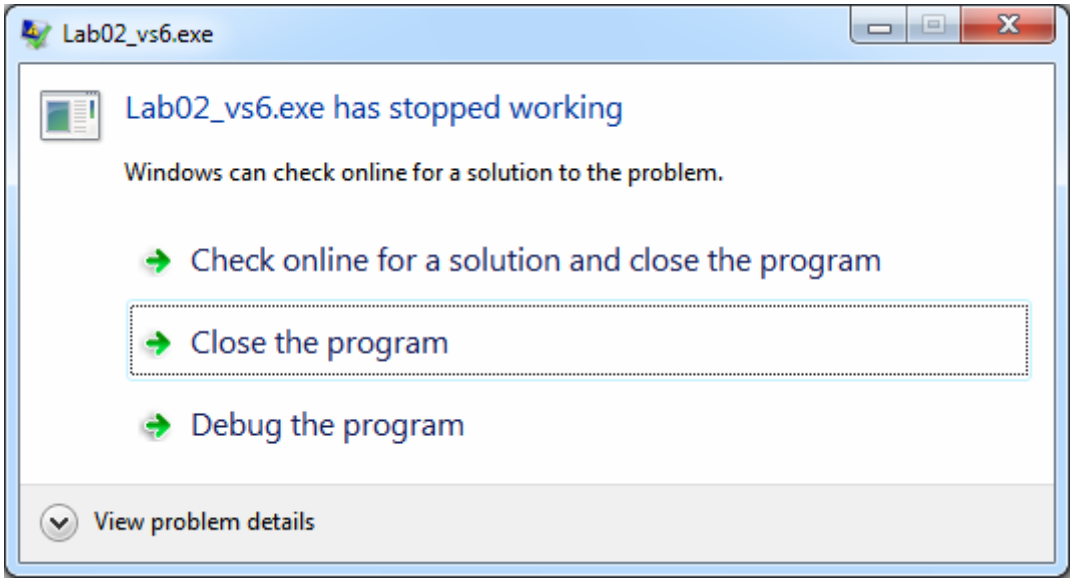
Pamięć alokowana dynamicznie w języku C. Wskaźniki

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

struct Student
{
	int nralb;
	char imie[100];
};

int main()
{
	struct Student * ptr;

	printf("sizeof(struct Student) == %d B\n", sizeof(struct Student));
	printf("sizeof(ptr) == %d B\n", sizeof(ptr));

	strcpy(ptr->imie, "Donald");
	ptr->nralb = 12345;

	return 0;
}
```

Uwaga! Powyższy kod jest niekompletny. Czego brakuje?

Zmienna `ptr` to zdecydowanie zbyt mało pamięci, aby zmieściło się tam 99-znakowe imię i liczba całkowita. Jest to tylko wskaźnik, czyli informacja o tym, gdzie znajdują się właściwie dane. Kompilator próbuje nas ostrzec przez próbą użycia niezainicjalizowanej zmiennej lokalnej:

```
warning C4700: local variable 'ptr' used without having been initialized
```



Prawdziwy problem jest jednak poważniejszy. Brakuje nam pamięci, w której moglibyśmy przechować potrzebne dane (np. imię).

© Krzysztof Urbański 2011

Dynamicznie można tę pamięć przydzielić wywołując funkcję malloc(…), a zwolnić za pomocą free(…). Ile bajtów chcemy zaalokować dynamicznie? Przynajmniej tyle, ile zajmuje jeden cały student, a tego z kolei dowiemy się używając **sizeof(struct Student).**

```
struct Student * ptr;
ptr = malloc(sizeof(struct Student));
if(ptr == NULL) exit(0); //awaria - brak pamięci?...
strcpy(ptr->imie, "Donald");
ptr->nralb = 12345;
printf("nralb: %d, imie: %s\n", ptr->nralb, ptr->imie);
free(ptr);
```

Możliwe jest alokowanie większych obszarów pamięci niż tylko miejsce na pojedynczego studenta, wystarczy wpisać odpowiednio większą liczbę bajtów w argumencie wywołania malloc(…):

```
struct Student * ptr;
ptr = malloc(10 * sizeof(struct Student));
for(i=0; i<10; i++)
{
     sprintf(ptr[i].imie, "Anonim %d", i + 1);
     ptr[i].nralb = 1000 + i;
}

for(i=0; i<10; i++)
     printf("%d. nralb: %d, imie: %s\n", i, ptr[i].nralb, ptr[i].imie);
```

Pozornie mamy niejednoznaczność: w obu przykładach użyłem identycznej zmiennej wskaźnikowej ptr. W pierwszym przypadku wskazywała ona na jednego studenta, w drugim – na ekipę 10 studentów.

Skąd wiadomo, ile pamięci znajduje się pod adresem wskazywanym przez wskaźnik? Odpowiedź jest przygnębiająca: nie wiadomo! Gdyby zajrzeć do wnętrza funkcji malloc(…) to moglibyśmy znaleźć okrężny sposób na uzyskanie tej wiedzy, ale założenie jest takie, że programista we własnym zakresie ma obowiązek pamiętać, ile pamięci zaalokował. Przydatna będzie tutaj np. dodatkowa zmienna, w której przechowamy tę informację:

```
int N = 10;
ptr = malloc(N * sizeof(struct Student));
…
for(i=0; i< N; i++)
     printf("%d. nralb: %d, imie: %s\n", i, ptr[i].nralb, ptr[i].imie);
…
free(ptr);
```

W różnych miejscach kodu odwołujemy się wtedy do tej zmiennej, a nie do magicznej liczby 10 czy dowolnej innej.

Zwróć uwagę na **ptr->nralb = 12345;** oraz **ptr[i].nralb = 12345;**. Dlaczego te zapisy można stosować zamiennie?

© Krzysztof Urbański 2011

Przede wszystkim warto wiedzieć, że ptr->nralb można inaczej zapisać jako (*ptr).nralb. Po drugie pamiętajmy, że tablice i wskaźniki w C zachowują się bardzo podobnie. Na tyle podobnie, że można używać operatora indeksowania [] zarówno dla tablic jak i dla wskaźników. Nazwa tablicy to nic innego jak adres miejsca w pamięci, gdzie został umieszczony element o indeksie 0. Ale wskaźnik to także adres miejsca w pamięci, gdzie przechowujemy dane.

```
int i;
int tab[5];
int *ptr = tab;

tab[0] = 10;
tab[1] = 20;
tab[2] = 30;

for(i=0; i<5; i++)
     printf("tab[%d] = %d\n", i, tab[i]);
```

Wyniki:

```
tab[0] = 10
tab[1] = 20
tab[2] = 30
tab[3] = -858993460
tab[4] = -858993460
```

Dopiszmy teraz kilka instrukcji do powyższego kodu:

```
ptr[0] = 0; //albo *ptr = 0;
ptr[3] = 3;
ptr[4] = 4;
for(i=0; i<5; i++)
     printf("tab[%d] = %d\n", i, tab[i]);
```

Wyniki:

```
tab[0] = 0
tab[1] = 20
tab[2] = 30
tab[3] = 3
tab[4] = 4
```

Dlaczego modyfikacje zmiennej ptr wpłynęły na zawartość tablicy tab? Dlatego, że zarówno tab, jak i ptr wskazują na dokładnie ten sam obszar (adres) pamięci.

© Krzysztof Urbański 2011

Arytmetyka wskaźników, czyli elementy magii stosowanej

Wskaźniki to coś więcej niż adresy, stąd nie używamy określenia „adres w pamięci" a „wskaźnik na obszar pamięci". Wskaźniki są inteligentniejsze niż adresy, a główna różnica polega na tym, że wskaźnik „wie" jak należy interpretować dane, na które wskazuje (posiada określony typ).

Weźmy dwa wskaźniki ptr_c oraz ptr_i:

```
char tab[40];
char *ptr_c = tab;
int *ptr_i = (int*)tab;//uwaga! Jawne rzutowanie typów!
```

W pewnym miejscu w pamięci (pod pewnym adresem) znajduje się tablica o nazwie tab. Zajmuje ona 40 bajtów. Oba wskaźniki zostały tak spreparowane, że wskazują na dokładnie ten sam obszar pamięci gdzie znajduje się tablica. Wykonajmy kilka sztuczek:

```
char tab[40];
char *ptr_c = tab;
int *ptr_i = (int*)tab;
int x;

tab[0] = 65;
tab[1] = 66;
tab[2] = 'C';
ptr_c[3] = 'r';
ptr_c[4] = 's';
for(x=0; x<7; x++)
     printf("tab[%d]: %d %c\n", x, tab[x], tab[x]);
```

Wyniki:

```
tab[0]: 65 A
tab[1]: 66 B
tab[2]: 67 C
tab[3]: 114 r
tab[4]: 115 s
tab[5]: -52 ╠
tab[6]: -52 ╠
```

© Krzysztof Urbański 2011

Na razie wszystko wygląda przewidywalnie.

Ale weźmy teraz:

`*ptr_i = 512;` albo `ptr_i[0] = 512;`

czyli pod miejsce w pamięci wskazywane przez wskaźnik ptr_i wpiszmy liczbę całkowitą 512, traktując to miejsce jak liczbę całkowitą 32-bitową.

```
tab[0]: 0
tab[1]: 2 ☻
tab[2]: 0
tab[3]: 0
tab[4]: 115 s
tab[5]: -52 ├
tab[6]: -52 ├
```

Zmianie uległy aż 4 bajty w pamięci, na pozycji o indeksie 1 pojawiła się wesoła buźka (znak o wartości 2). Zadanie do przemyślenia: dlaczego akurat 2 i co stało się z liczbą 512?

Idąc za ciosem, wpiszmy w to samo miejsce tablicy wartość 1 (druga wesoła buźka, tym razem biała a nie czarna) i wyświetlmy zawartość tablicy tab[] oraz sprawdźmy, co znajduje się w miejscu wskazywanym przez ptr_i:

```
tab[0]: 0
tab[1]: 1 ☺
tab[2]: 0
tab[3]: 0
ptr_i[0] == *ptr_i == 256
```

Zadania do realizacji:

1. Jaka wartość się pojawi w ptr_i[0], kiedy wpiszemy pod pierwsze 4 pozycje tab[] czarne buźki? A jaka, gdy będą to białe buźki?
2. Co należy wpisać do ptr_i[0], aby w tab[] pojawiły się buźki czarne i białe: ☻ ☺ ☻ ☺?

Sprawdź równoważność zapisów:

`*ptr_i`, `*(ptr_i+0)` oraz `ptr_i[0];`
`*(ptr_i+1)` oraz `ptr_i[1];`
`*(ptr_i+2)` oraz `ptr_i[2];`

To właśnie nazywamy arytmetyką wskaźników. Dodanie wartości liczbowej do wskaźnika przesuwa jego wskazanie o wielokrotność bajtów, na które wskazuje. Jeśli mamy do czynienia ze wskaźnikiem char*, to typ docelowy char zajmuje 1 bajt. Zwiększenie wskaźnika o 1 przesunie nas do kolejnego elementu, czyli o 1 bajt dalej. Kiedy jednak mamy do czynienia ze wskaźnikiem int*, to docelowy typ (a konkretnie int) zajmuje 4 bajty, więc zwiększenie wskaźnika o 1 jest równoważne

© Krzysztof Urbański 2011

przesunięciu adresu o 4 bajty, czyli w taki sposób, aby wskazywany był element leżący o 1 pozycję dalej. Wyższość wskaźników nad surowymi adresami polega m.in. na tym, że w przypadku wskaźników to kompilator będzie 'pilnował', aby automatycznie przeliczyć na bajty to logiczne przesunięcie.

Dodawanie/odejmowanie wartości liczbowych do wskaźnika jest równoważne przesuwaniu tego wskaźnika dokładnie tak, jak to się dzieje w przypadku tablic i operatora indeksowania – czyli przesuwanie się o pewną liczbę elementów.

Wracając do naszej struktury Student: kiedy już mamy wskaźnik ptr do niej i mamy jednocześnie pewność że obszar pamięci, na który ten wskaźnik wskazuje, jest na tyle pojemny, że zmieścimy w nim przykładowo 5 studentów:

```
struct Student *ptr = malloc(5 * sizeof(struct Student));
```

Lepiej jest używać calloc, który tym się różni od malloc, że automatycznie zeruje alokowaną pamięć (czyli nie będą się w niej znajdowały przypadkowe śmieci), oraz nieco inaczej przekazuje się do niego informację o liczbie bajtów do zaalokowania:

```
struct Student *ptr = calloc(5, sizeof(struct Student)); //5 elementów po sizeof(…) bajtów każdy
```

```
ptr->nralb = 1000;              //pierwszy student
(*ptr).nralb = 1000;            //też pierwszy student
(*(ptr+0)).nralb = 1000;        //tutaj też pierwszy student
(*(ptr+1)).nralb = 1001;        // drugi student (ten za pierwszym)
(ptr+2)->nralb = 1002;          //trzeci
ptr[3].nralb = 1003;            //czwarty
(&ptr[4])->nralb = 1004;        //piąty
```

Wyniki:

```
0. nralb: 1000, imie:
1. nralb: 1001, imie:
2. nralb: 1002, imie:
3. nralb: 1003, imie:
4. nralb: 1004, imie:
```

Zadania:

1. Wczytaj liczbę N z klawiatury, która będzie określała pojemność bazy studentów. Pamiętaj o kontroli poprawności wprowadzonych danych (np. wpisanie ALA, 0, 100000000000000000000 albo -2000 powinno być sygnalizowane ostrzeżeniem o błędzie i ponowieniem pytania o wprowadzenie liczby)

© Krzysztof Urbański 2011

2. Zaalokuj pamięć używając malloc albo calloc. Jeśli używasz malloc, to dodatkowo ręcznie zainicjalizuj pamięć (najlepiej wypełnij zerami).
3. Przygotuj następujące menu wyboru:
   (1) wyświetl wszystkie rekordy w konsoli
   (2) wprowadź jeden rekord z konsoli do pamięci
   (3) zapisz dane do pliku tekstowego 'baza.txt'
   (4) zapisz dane do pliku binarnego 'baza.bin'
   (5) wczytaj dane z pliku tekstowego 'baza.txt'
   (6) wczytaj dane z pliku binarnego 'baza.bin'
   (0) koniec
4. W przypadku opcji (2) powinna pojawić się prośba o podanie numeru pozycji w tablicy, gdzie zostaną wczytane dane. Ta pozycja to liczba z zakresu [0..N-1]. Jeśli ktoś wpisze błędną wartość, należy ponowić pytanie. Z klawiatury wczytane będą nralb oraz imię, tutaj też warto zadbać o kontrolę poprawności danych (np. nie wolno wczytać więcej niż 99 znaków do pola imie).
5. Jeśli chodzi o pliki, to wykonuj punkty w takiej kolejności, w jakiej zostały zapisane opcje. Pliki binarne wymagają otwarcia do zapisu i odczytu w trybie „b". Zapis i odczyt danych w plikach binarnych najlepiej zrealizować funkcjami fwrite(…) oraz fread(…). Opcja (6) jest najtrudniejsza i wymaga konsultacji z prowadzącym zajęcia. Główny problem to możliwość zmiany liczby elementów bazy w kolejnych uruchomieniach aplikacji.
   - Scenariusz A: użytkownik uruchamia aplikację, wybiera 5 elementów, wpisuje dane, zapisuje wszystko na dysk i kończy aplikację. W drugim podejściu uruchamia aplikację, wpisuje 3 rekordy i wczytuje dane z dysku. Na dysku było zapisane 5 rekordów, obecnie w pamięci jest miejsce na 3, dane się nie mieszczą, następuje koniec świata.
   - Scenariusz B: podobnie jak poprzednio, ale na odwrót – zapisane na dysku są 3 rekordy, program oczekuje że uda się wczytać 5, część pamięci pozostaje niewypełniona danymi. Końca świata nie ma, ale może wystąpić mały kryzys.
   - Scenariusz C: Programista 'na sztywno' ustawia N na stałą wartość, np. 10 – na początek wystarczy, ale w przyszłości trzeba będzie coś z tym zrobić.
   - Scenariusz D: Aplikacja jest na tyle cwana, że sprawdza na początku, czy już istnieje na dysku plik z danymi. Jeśli tak, to liczy ile rekordów się w nim znajduje (załóżmy że 5). Użytkownik nie może w takiej sytuacji wybrać bazy 3-elementowej, no chyba że się mocno uprze. Rekordy są wczytywane z dysku do pamięci pojedynczo, aż do momentu, kiedy osiągnięty zostanie koniec pamięci bądź koniec dysku. Inny wariant to zwalnianie pamięci przed wczytaniem danych dysku i alokowanie dokładnie tyle, ile jest potrzebne (ile wynika z rozmiaru pliku na dysku).

© Krzysztof Urbański 2011