Podstawy Programowania 2

Wskaźniki i zmienne dynamiczne

Arkadiusz Chrobot

Katedra Systemów Informatycznych

5 marca 2024

Plan

1. Wskaźniki na wskaźniki
2. Wskaźniki na funkcje
3. Zmienne dynamiczne
4. Jak czytać skomplikowane deklaracje?

Wskaźniki na wskaźniki

Wskaźnik to zmienna, która może przechowywać *adres* innej zmiennej. W języku C możliwe jest zadeklarowanie wskaźnika, który może wskazywać na inny wskaźnik. Taki wskaźnik nazywa się *wskaźnikiem na wskaźnik* lub *podwójnym wskaźnikiem*. Jest on deklarowany według następującego schematu:

```
typ_danych **pointer_to_pointer;
```

Podwójny asterysk oznacza, że jest to wskaźnik mogący przechowywać adres innego wskaźnika. Jeśli w tej deklaracji występowałaby dodatkowa gwiazdka, to byłby to wskaźnik na wskaźnik na wskaźnik. Liczba gwiazdek określa *poziom* (ang. *level*) wskaźnika lub *poziom pośredniości* (ang. *indirection level*). Standard języka C określa, że kompilatory muszą obsługiwać przynajmniej do 12 poziomów pośredniości, ale w większości programów używane są wskaźniki jedno i dwu poziomowe.

Podwójne wskaźniki mogą być zmiennymi globalnymi, lokalnymi i parametrami funkcji.

Wskaźniki na wskaźniki

Przykład

```c
#include<stdio.h>

void display(int **pointer)
{
    printf("Adres we wskaźniku na wskaźnik: %p\n",pointer);
    printf("Adres we wskaźniku: %p\n",*pointer);
    printf("Wartość w zmiennej: %d\n",**pointer);
}
```

Wskaźniki na wskaźniki

Przykład

```
int main(void)
{
    int first_variable = 5;
    int second_variable = 6;
    int *first_pointer = &first_variable;
    int *second_pointer = &second_variable;
    int **pointer_to_pointer = &first_pointer;
    display(pointer_to_pointer);
    pointer_to_pointer = &second_pointer;
    display(pointer_to_pointer);
    return 0;
}
```

Wskaźniki na wskaźniki

Przykład – komentarz

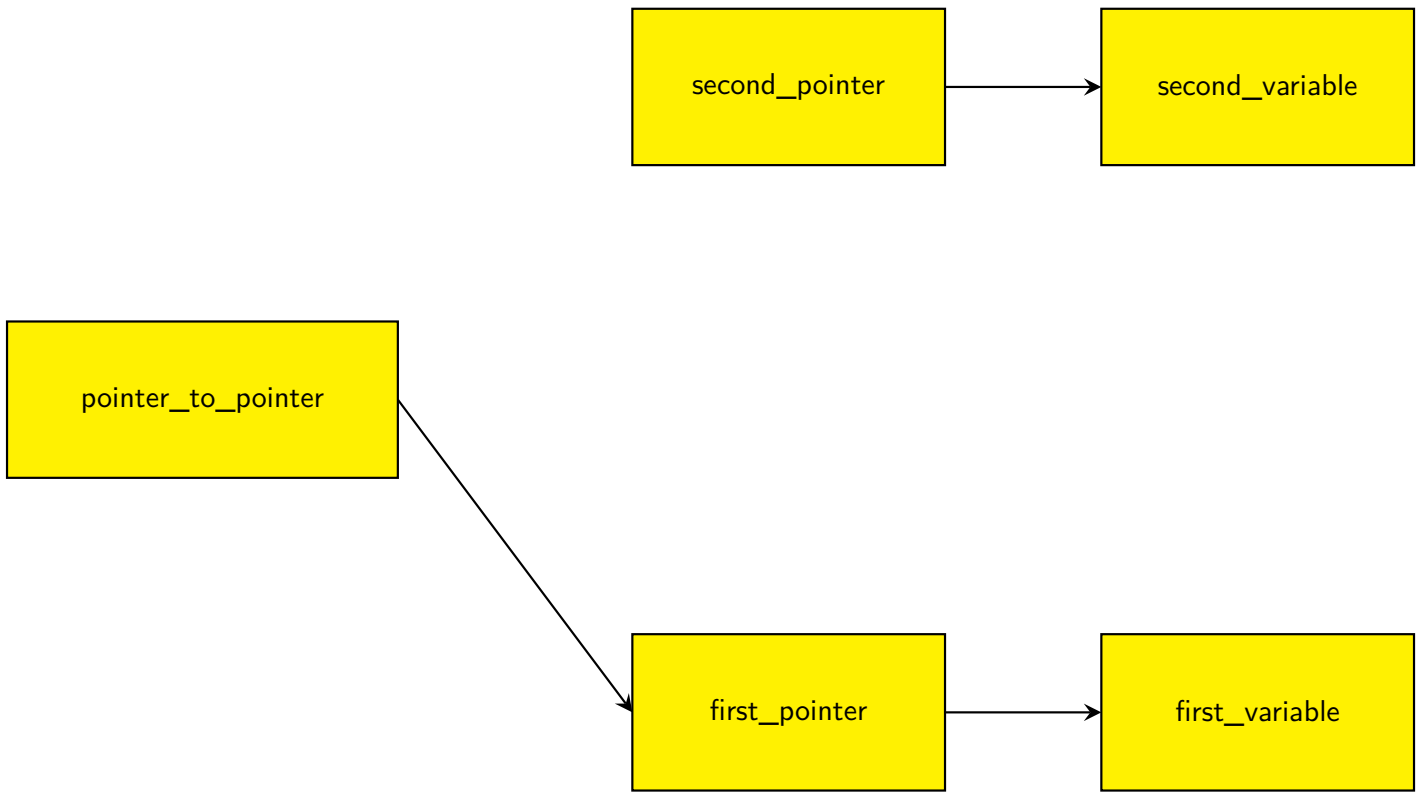
Program z dwóch poprzednich slajdów pokazuje jak działają podwójne wskaźniki. Funkcja `display()` ma argument będący wskaźnikiem na wskaźnik i w związku z tym przyjmuje argumenty tego samego typu. Najpierw wypisuje ona adres przechowywany w tym parametrze. Jest to adres innego wskaźnika. Następnie wyświetla adres umieszczony w tym innym wskaźniku. Aby otrzymać tę wartość jednokrotnie użyto na parametrze operatora dereferencji. Na koniec funkcja dwukrotnie stosuje ten operator, aby uzyskać wartość zapisaną w zmiennej wskazywanej przez wskaźnik, wskazywany przez parametr.

W funkcji `main()` są zadeklarowane dwie zmienne typu `int` oraz dwa wskaźniki tego samego typu. Zmienne są inicjowane, kolejno, liczbami 5 i 6, a wskaźniki adresami tych zmiennych. Potem deklarowany jest wskaźnik na wskaźnik, który inicjowany jest adresem pierwszego wskaźnika i wywoływana jest funkcja `display()`, której argumentem jest ten podwójny wskaźnik.

Wskaźniki na wskaźniki

Przykład – komentarz

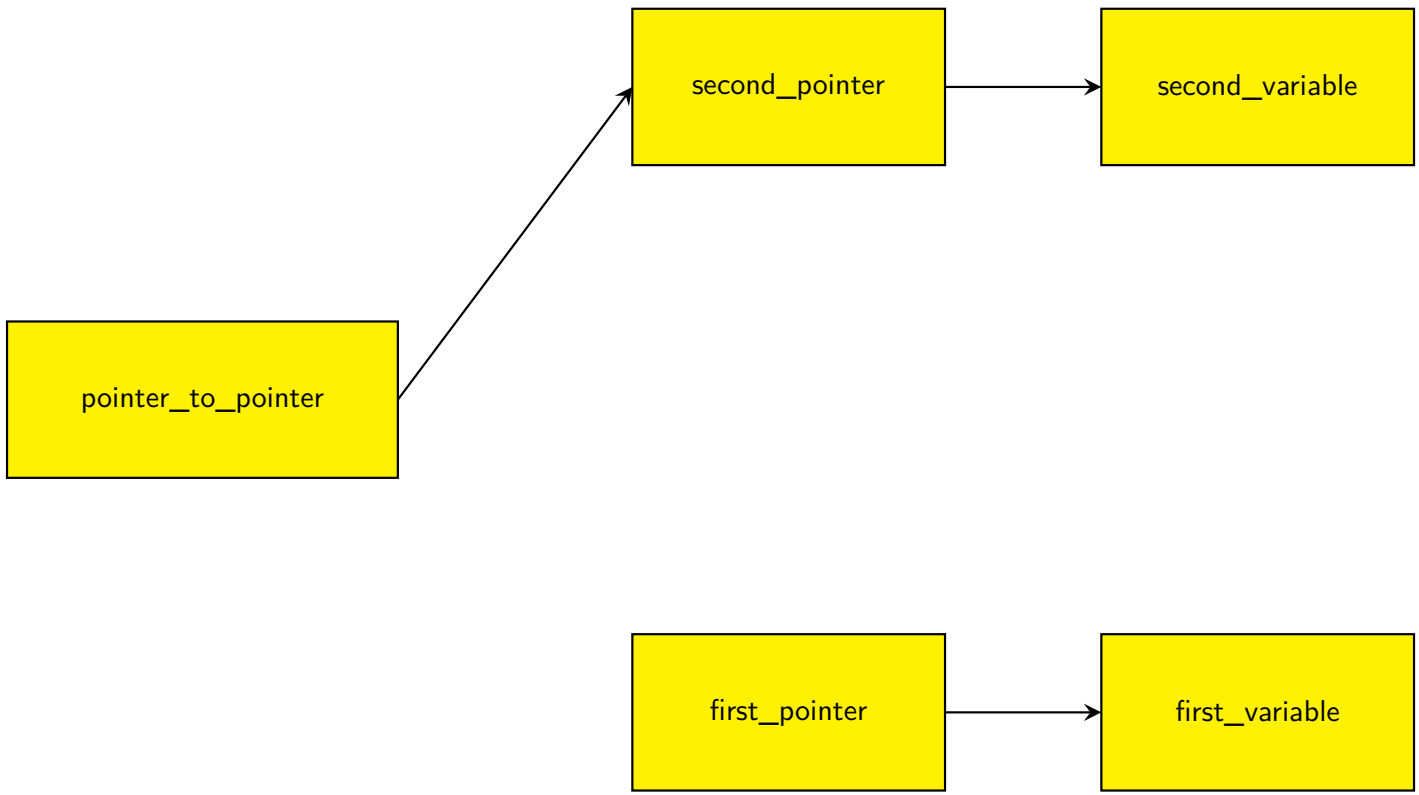
Następnie do wskaźnika na wskaźnik przypisywany jest adres drugiego wskaźnika i ponownie wywoływana jest funkcja `display()`. Tym razem wypisuje ona na ekranie inne dane. Rysunek na dole slajdu ilustruje działanie podwójnego wskaźnika w opisywanym programie.



Wskaźniki na wskaźniki

Przykład – komentarz

Następnie do wskaźnika na wskaźnik przypisywany jest adres drugiego wskaźnika i ponownie wywoływana jest funkcja `display()`. Tym razem wypisuje ona na ekranie inne dane. Rysunek na dole slajdu ilustruje działanie podwójnego wskaźnika w opisywanym programie.



Wskaźniki na funkcje

Funkcje tak samo jak dane są umieszczane w komórkach pamięci operacyjnej komputera. Zatem, tak jak zwykłe dane, mogą one być wskazywane przez wskaźniki. Za pomocą tych zmiennych wskaźnikowych funkcje, nawet te, które przyjmują argumenty wywołania, mogą być wywoływane. Wskaźnik na funkcje musi mieć określony typ, aby można było nim wskazywać i wywoływać funkcje. Przykładowo, jeśli funkcja nie zwraca żadnej wartości (inaczej: zwraca `void`) oraz nie przyjmuje żadnych argumentów wywołania, to wskaźnik na taką funkcję powinien być zadeklarowany następująco:

```
void (*function_pointer)(void);
```

Proszę zwrócić uwagę na użycie nawiasów okrągłych. Bez nich otrzymalibyśmy nie wskaźnik na funkcję, ale prototyp (nagłówek) funkcji, która nie przyjmuje żadnych argumentów wywołania, a zwraca wskaźnik nieokreślonego typu. Jeśli chcielibyśmy zadeklarować wskaźnik na funkcję, która przyjmuje dwa argumenty wywołania typu `int` i zwraca wartość tego samego typu, to moglibyśmy to zrobić następująco:

```
int (*another_function_pointer)(int, int);
```

Wskaźniki na funkcje

Deklaracje wskaźników na funkcje mogą być jeszcze bardziej skomplikowane, do czego wrócimy pod koniec wykładu. Dodatkowo możemy tworzyć struktury zawierające wskaźniki na funkcje, lub tablice, których elementy są takimi wskaźnikami. Następne slajdy zawierają dwa przykłady programów ilustrujących użycie wskaźników o takich samych typach, jak wskaźniki opisane na poprzednim slajdzie.

Wskaźniki na funkcje

Przykład — prosty wskaźnik na funkcję

```c
#include<stdio.h>

void say_hello(void)
{
    puts("Cześć!");
}

int main(void)
{
    void (*function_pointer)(void) = 0;
    function_pointer = say_hello;
    function_pointer();
    return 0;
}
```

Wskaźniki na funkcje

Przykład — komentarz

W kodzie źródłowym zaprezentowanego na poprzednim slajdzie programu umieszczona jest funkcja `say_hello()`, która nie przyjmuje żadnych argumentów wywołania, ani nie zwraca żadnej wartości. W funkcji `main()` tego programu zadeklarowano wskaźnik na taką funkcję. Jest to wskaźnik lokalny, dlatego w miejscu jego deklaracji nadano mu wartość `0`. Brak inicjacji takiego wskaźnika nie jest sygnalizowany przez kompilator, ale mógłby mieć poważne konsekwencje, stąd lepiej wykonać taką operację. W następnym wierszu umieszczona jest na pozór dosyć zagadkowa operacja przypisania. Dosłownie można ją zinterpretować jako przypisanie do wskaźnika nazwy funkcji. W rzeczywistości, nazwa funkcji jest traktowana w języku C jako wskaźnik, a więc w tym wierszu do wskaźnika na funkcję zapisywany jest adres tej funkcji. Kod programu można trochę skrócić zastępując oba opisywane wiersze pojedynczą instrukcją przypisania:

```
void (*function_pointer)(void) = say_hello;
```

Wskaźniki na funkcje

Przykład — komentarz c.d.

Instrukcja zawarta w kolejnym wierszu wygląda jak wywołanie funkcji, ale zamiast jej nazwy jest użyta nazwa wskaźnika. Faktycznie, w tym miejscu jest wywoływana funkcja, nie bezpośrednio, ale za pomocą zmiennej wskaźnikowej, która na nią wskazuje. Para nawiasów okrągłych, czyli `()`, znajdujących się za nazwą tego wskaźnika, to *operator wywołania funkcji*, który nakazuje jej uruchomienie w miejscu, w którym występuje. Jeśli funkcja przyjmowałaby argumenty wywołania, to ich lista znalazłaby się między tymi nawisami, co obrazuje następny program.

Wskaźniki na funkcje

Przykład — bardziej skomplikowany wskaźnik na funkcję

```c
#include<stdio.h>

int add_up(int first, int second)
{
    return first+second;
}

int main(void)
{
    int (*another_function_pointer)(int, int) = NULL;
    another_function_pointer = add_up;
    printf("Po dodaniu %d do %d otrzymamy %d.\n",3,2,
                another_function_pointer(3,2));
    return 0;
}
```

Wskaźniki na funkcje

Przykład — komentarz

W programie zamieszczonym na poprzednim slajdzie mamy bardziej skomplikowaną funkcję, która dodaje wartości swoich parametrów i zwraca ich sumę. W funkcji `main()` jest zadeklarowany i zainicjowany wskaźnik na taką funkcję. Tym razem nadano mu wartość stałej `NULL`, aby pokazać, że ona także może być w takiej sytuacji użyta. W kolejnym wierszu do tego wskaźnika zapisywany jest adres funkcji `add_up()`. Podobnie jak w poprzednim programie te dwa wiersze możemy zastąpić pojedynczym:

```
int (*another_function_pointer)(int, int) = add_up;
```

Ponieważ funkcja `add_up()` ma dwa parametry, to w miejscu jej wywołania musimy umieścić argumenty, które zostaną za te parametry podstawione. To samo dotyczy sytuacji, w której jest ona wywoływana za pośrednictwem wskaźnika. Przekazujemy jej jako argumenty dwie liczby: 3 i 2. Dodatkowo proszę zwrócić uwagę na to, że wartość zwracana przez tę funkcję jest argumentem funkcji `printf()`, która wypisze wynik jej działania na ekran.

Wskaźniki na funkcje

Zastosowanie w standardowej bibliotece języka C

Wskaźniki na funkcje są dosyć często używane w funkcjach dostępnych w standardowej bibliotece języka C. Na przykład jednym z parametrów funkcji `qsort()` sortującej liniową tablicę przy użyciu algorytmu Quick Sort (bardzo wydajny algorytm sortowania tablic) jest właśnie taki wskaźnik. Argumentem podstawianym za ten parametr jest adres funkcji, która porównuje wartości dwóch elementów sortowanej tablicy. Zaprezentowany na następnych slajdach program demonstruje sposób użycia `qsort()` do sortowania jednowymiarowej tablicy liczb całkowitych.

Wskaźniki na funkcje

Zastosowanie funkcji qsort()

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

#define LENGTH 20

void populate(int array[], unsigned int length)
{
    srand(time(0));
    for(int i=0; i<length; i++)
        array[i]=-10+rand()%21;
}
```

Wskaźniki na funkcje

Zastosowanie funkcji `qsort()` — komentarz

W programie włączone są trzy pliki nagłówkowe. Pierwszy, czyli `stdio.h` jest użyty, ponieważ program korzysta z funkcji `printf()`. Dwa kolejne zostały włączone ze względu na zastosowanie w programie generatora liczb pseudolosowych. Jest jednak jeszcze jeden powód użycia `stdlib.h` — w tym pliku nagłówkowym jest zadeklarowana funkcja `qsort()`.
Funkcja `populate()` wypełnia przekazaną jej jako pierwszy argument tablicę liczbami całkowitymi losowanymi z przedziału $[-10, 10]$. Drugim argumentem przekazywanym do funkcji jest liczba elementów w tablicy.

Wskaźniki na funkcje

Zastosowanie funkcji qsort()

```
void print(int array[], unsigned int length)
{
    for(int i=0; i<length; i++)
        printf("%d ",array[i]);
    puts("");
}

int compare(const void *first, const void *second)
{
    return *(int *)first - *(int *)second;
}
```

Wskaźniki na funkcje

Zastosowanie funkcji `qsort()` — komentarz

Funkcja `print()` wypisuje wartości z tablicy, przekazanej jej jako pierwszy argument, na ekranie. Jako drugi argument do tej funkcji jest przekazywana liczba elementów w tej tablicy.
Funkcja `compare()` będzie wywoływana przez `qsort()` celem porównania wartości dwóch elementów tablicy. Powinna ona zwrócić ujemną liczbę całkowitą, jeśli wartość pierwszego elementu jest mniejsza niż drugiego, dodatnią liczbę całkowitą, jeśli jest odwrotnie lub zero, jeśli obie wartości są równe.
Aby móc sortować tablicę elementów dowolnego typu, funkcja `qsort()` otrzymuje ją przez parametr będący wskaźnikiem typu `void *`. Zatem wywołując funkcję `compare()` może jej przekazać jedynie wskaźniki tego samego typu na dwa elementy tej tablicy, których wartości należy porównać.

Wskaźniki na funkcje

Zastosowanie funkcji `qsort()` — komentarz

Funkcja `compare()` rzutuje te wskaźniki na typ `int *`, a następnie wykonuje ich dereferencję i odejmuje od siebie liczby zawarte w elementach tablicy wskazywanych przez nie. Wynikiem tego działania będzie liczba oczekiwana przez `qsort()`, dlatego jest zwracana przez `compare()`.

Wskaźniki na funkcje

Zastosowanie funkcji qsort()

```
int main(void)
{
    int numbers[LENGTH];
    populate(numbers,LENGTH);
    print(numbers,LENGTH);
    qsort(numbers,LENGTH,sizeof(numbers[0]),compare);
    print(numbers,LENGTH);
    return 0;
}
```

Wskaźniki na funkcje

Zastosowanie funkcji `qsort()` — komentarz

W funkcji `main()` zadeklarowana jest tablica o 20 elementach, która następnie wypełniana jest pseudolosowymi liczbami całkowitymi. Potem te liczby z tablicy wypisywane są na ekranie i wywoływana jest funkcja `qsort()` celem posortowania tej struktury danych. Jako pierwszy argument przekazywana jest je tablica, którą należy posortować, a jako drugi liczba elementów tej tablicy. Trzecim argumentem jest rozmiar pojedynczego elementu. Ostatni argument funkcji `qsort()` to adres `compare()`. Jak wspomniano wcześniej nazwa funkcji, jest jednocześnie wskaźnikiem zawierającym jej adres, dlatego w tym programie została użyta w charakterze czwartego argumentu `qsort()`.

Funkcje, które są przy pomocy wskaźników wywoływane przez inne funkcje są określane są jako *funkcje wywołań zwrotnych*. W opisywanym programie taką funkcją jest `compare()`, wywoływane przez `qsort()`, wtedy, kiedy ta ostatnia musi porównać wartości dwóch elementów tablicy.

Zmienne dynamiczne

Wskaźniki pełnią w programowaniu jeszcze jedną, bardzo ważną rolę — pozwalają zrealizować koncepcję tzw. zmiennych dynamicznych. Czym są takie zmienne? Zanim odpowiemy na to pytanie, przypomnijmy jakie są główne rodzaje zasięgu zmiennych i co o nich najważniejszego wiemy:

- *zmienne globalne* — są to zmienne tworzone w obszarze danych globalnych programu w trakcie jego uruchamiania. Są one inicjowane wartościami domyślnymi dla ich typu i istnieją przez cały czas wykonywania programu, a więc są niszczone wtedy gdy on się zakończy.
- *zmienne lokalne* — nazywane także zmiennymi *automatycznymi*. Związane są one z funkcjami i tworzone są w momencie ich uruchamiania (wywoływania) w obszarze pamięci, który nazywa się stosem. Są one częścią ramki stosu związanej z wywoływaną funkcją. Taką ramkę nazywa się także rekordem aktywacyjnym. Zmienne lokalne nie są inicjowane, mają przypadkową wartość początkową. Są one niszczone w momencie zakończenia wykonania funkcji, w której są zadeklarowane. Ich widoczności jest zależna od bloku kodu, w którym zostały zadeklarowane.

Zmienne dynamiczne

Zmienne dynamiczne mają charakterystykę plasującą je między dwoma opisanymi na poprzednim slajdzie rodzajami zmiennych. O ich powstaniu i usunięciu, czyli czasie życia, a także o ich widoczności decyduje programista. Stąd bierze się ich nazwa — powstają one i są usuwane w trakcie wykonywania programu. Zmienne te tworzone są w obszarze pamięci programu, który nazywamy *stertą* (ang. *heap*). Do ich tworzenia i niszczenia służą specjalne podprogramy, które są częścią języka programowania. W przypadku języka C są to funkcje, które będą opisane na następnych slajdach. Podprogramy tworzące zmienne dynamiczne pozwalają określić programiście rozmiar pamięci, czyli liczbę komórek, które będą tworzyły taką zmienną, ale nie pozwalają nadać jej nazwy. Za to zwracają one adres nowej zmiennej, który można zapisać we wskaźniku. W ten sposób wskaźnik staje się jedynym łącznikiem między zmienną dynamiczną, a resztą programu. Zmienna dynamiczna po takim przypisaniu staje się zmienną wskazywaną. Dodatkowo wskaźnik, jeśli ma określony typ, determinuje również typ zmiennej dynamicznej.

Zmienne dynamiczne

Możliwe jest wskazywanie kilkoma wskaźnikami tej samej zmiennej dynamicznej i tym samym interpretowanie na różne sposoby informacji w niej zapisanej, ale ten przypadek jest dosyć skomplikowany i nie będzie na tym wykładzie szerzej komentowany. Czynność tworzenia zmiennej dynamicznej sprowadza się do rezerwacji dla niej pewnego ciągłego obszaru na stercie i nazywa się przydziałem lub alokacją pamięci. Wbrew pozorom nie jest to prosta praca i nie zawsze musi zakończyć się sukcesem. Sposób jej wykonania zależny jest od budowy komputera i systemu operacyjnego, pod którego kontrolą wykonywany jest program. Szczegóły działania podprogramów alokujących pamięć na stercie nie będą nas interesowały w ramach tego wykładu. Musimy jednak pamiętać, aby przed zakończeniem programu, lub wtedy gdy dane zmienne dynamiczne przestaną być przydatne w programie, zwolnić przydzieloną na nie pamięć. Do tej czynności służą inne podprogramy, które oznaczają przydzieloną na zmienną pamięć, jako wolną, czyli możliwą do wykorzystania w kolejnych przydziałach pamięci na stercie. Czynność zwalniania pamięci nazywa się także dealokacją i jest ona równoznaczna usunięciu (zniszczeniu) zmiennej dynamicznej.

Zmienne dynamiczne

Funkcje w języku C do obsługi sterty

W języku C istnieją cztery funkcje odpowiedzialne za zarządzanie (przydział i zwalnianie) pamięcią na stercie. Opisy tych funkcji znajdują się w tabelach na tym i kolejnych slajdach.

Nazwa funkcji	**Opis**
`malloc()`	Funkcja pobiera jeden argument, którym jest wyrażenie określające rozmiar pamięci (w bajtach), która ma być przydzielona na stercie. Zwracana przez nią wartość jest typu `void *` i jest to adres pierwszej komórki należącej do przydzielonego obszaru pamięci, który krótko będziemy nazywać adresem lub wskaźnikiem przydzielonej pamięci lub zmiennej dynamicznej. Jeśli przydział się nie powiedzie, to funkcja zwraca `NULL`. Przydzielony obszar pamięci jest niezainicjowany.

Zmienne dynamiczne

Funkcje w języku C do obsługi sterty

Nazwa funkcji	**Opis**
`calloc()`	Właściwie jest to odmiana funkcji `malloc()`, która została zaprojektowana, aby ułatwić przydzielanie pamięci na tablice. Przyjmuje ona dwa argumenty wywołania. Pierwszy określa liczbę elementów tworzonej tablicy dynamicznej, a drugi rozmiar pojedynczego elementu. Utworzona w ten sposób tablica jest zainicjowana wartościami zerowymi.

Zmienne dynamiczne

Funkcje w języku C do obsługi sterty

Nazwa funkcji	Opis
`free()`	Ta funkcja odpowiedzialna jest za zwolnienie pamięci. Nie zwraca ona żadnej wartości, ale przyjmuje wskaźnik na pamięć do zwolnienia. Ta pamięć musi być wcześniej przydzielona przez jedną z trzech funkcji, które do tego służą, inaczej działanie programu może się zakończyć poważnym błędem. Jeśli przekazany jej wskaźnik będzie miał wartość `NULL`, to funkcja nie wykona żadnej czynności. Należy pamiętać, że funkcja nie zeruje obszaru pamięci, który zwalnia, jedynie oznacza go jako wolny. Dane, które były w nim przechowywane nadal tam są, ale nie wolno się do nich już odwoływać. Funkcja nie zeruje również przekazanego jej wskaźnika i dopóki nie zapiszemy w nim nowego adresu nie należy się nim posługiwać. Żargonową nazwą takiego wskaźnika jest „wiszący wskaźnik".

Zmienne dynamiczne

Funkcje w języku C do obsługi sterty

Nazwa funkcji	Opis
`realloc()`	Ta funkcja dokonuje zmiany rozmiaru zaalokowanego obszaru sterty. Przyjmuje dwa argumenty. Pierwszym jest wskaźnik do przydzielonego już obszaru pamięci, a drugim jego nowy rozmiar wyrażony w bajtach. Funkcja ta ma typ wartości zwracanej `void *` i zwraca adres obszaru pamięci, jeśli udało się zmienić jego wielkość lub `NULL` jeśli ta operacja zakończyła się niepowodzeniem. Zwrócony adres może być różny od adresu zapisanego w przekazanym wskaźniku, gdyż funkcja może w przypadku rozszerzania obszaru napotkać problemy z powiększeniem bieżącego miejsca i przekopiować dane do nowego bloku pamięci. Jeśli obszar pamięci jest poszerzany, to dane, które były w nim zgromadzone są zachowywane w całości. Jeśli jest on pomniejszany, to część z nich może ulec usunięciu. Jeśli przekazany do funkcji wskaźnik będzie miał wartość `NULL`, to zachowa się ona jak `malloc()`, a jeśli przekazany jej rozmiar będzie miał wartość `0`, to zachowa się ona jak `free()`.

Zmienne dynamiczne

Wszystkie opisane w tabelach funkcje są zadeklarowane w pliku nagłówkowym `stdlib.h`. Z kolei w pliku `string.h` umieszczono deklaracje dwóch funkcji, które mogą być pomocne w zarządzaniu zmiennymi dynamicznymi. Z pierwszą już się spotkaliśmy — to `memset()`. Zapisuje ona określoną wartość w każdym bajcie wskazanego jej obszaru pamięci. Funkcja ta przyjmuje trzy argumenty wywołania. Pierwszym jest wskaźnik (typu `void *`) na obszar pamięci, który ma być zapisany, drugim jest wartość (typu `int`), która ma być umieszczona w jego poszczególnych komórkach, a trzecim jest rozmiar tego obszaru w bajtach. Funkcja `memset()` zwraca wskaźnik typu `void *` na wypełniony obszar pamięci. Druga funkcja to `memcpy()`, która kopiuje zawartość jednego obszaru pamięci do drugiego. Przyjmuje ona trzy argumenty wywołania. Dwa pierwsze to wskaźniki typu `void *` na odpowiednio obszar docelowy i źródłowy pamięci. Trzeci argument to liczba bajtów do skopiowania. Funkcja zwraca wskaźnik (typu `void *`) na obszar docelowy.

Zmienne dynamiczne

Przykład — zmienna dynamiczna typu int

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

int main(void)
{
    int *variable = (int *)malloc(sizeof(int));
    if(variable) {
        printf("Adres zmiennej dynamicznej: %p\n",variable);
        *variable = 24;
        printf("Wartość zmiennej dynamicznej %d\n",*variable);
        free(variable);
        variable=NULL;
    }
    return 0;
}
```

Zmienne dynamiczne

Przykład — zmienna dynamiczna typu `int`

Na poprzednim slajdzie przedstawiono prosty program bez podziału na funkcje, w którym używana jest zmienna dynamiczna typu `int`. Jest ona tworzona za pomocą wywołania funkcji `malloc()`. Jej rozmiar jest określany za pomocą operatora `sizeof` zastosowanego dla typu `int`. Wartość (adres) zwracana przez tę funkcję jest rzutowana na typ `int *` i zapisywana do wskaźnika o nazwie `variable`. Następnie, po sprawdzeniu, czy przydział pamięci się powiódł, wypisywany jest na ekran adres zapisany w tej zmiennej wskaźnikowej. Następnie do zmiennej wskazywanej przez ten wskaźnik zapisywana jest liczba 24. W kolejnym wierszu wypisywana jest na ekran wartość tej zmiennej. Po wykonaniu wszystkich tych czynności zwalniana jest pamięć przeznaczona na zmienną dynamiczną przy pomocy `free()` i wskaźnikowi przypisywana jest wartość `NULL`. Żadna operacja nie może być wykonana na zmiennej dynamicznej dopóki nie upewnimy się, że została przydzielona na nią pamięć.

Zmienne dynamiczne

Przykład — dynamiczna tablica

Zaprezentowany program, który używa zmiennej dynamicznej typu `int` nie prezentuje pełni możliwości zmiennych tego rodzaju. Kolejny program tworzy tablicę o zmiennej liczbie elementów. Ta liczba nie jest znana przed uruchomieniem programu. Jest ona zwiększana w trakcie jego działania, aby umożliwić zapis do tablicy nowych wartości.

Zmienne dynamiczne

Przykład — dynamiczna tablica

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

Zmienne dynamiczne

Przykład — dynamiczna tablica

Z uwagi na to, że program będzie wypisywał komunikaty na ekranie, zarządzał pamięcią na stercie oraz używał generatora liczb pseudolosowych, zostały do niego włączone trzy pliki nagłówkowe: `stdio.h`, `stdlib.h` oraz `time.h`.

Zmienne dynamiczne

Przykład — dynamiczna tablica

```
int *add_element(int *array, int index, int value)
{
    static unsigned int length;
    const unsigned int DELTA = 10;
    if(index>=length) {
        length += DELTA;
        int *new_array = realloc(array, length * sizeof(int));
        if(new_array)
            array = new_array;
    }
    if(array)
        array[index]=value;
    return array;
}
```

Zmienne dynamiczne

Przykład — dynamiczna tablica

Funkcja `add_element()` jest, trochę wbrew nazwie, odpowiedzialna za dodanie nowej wartości do tablicy. Pobiera ona trzy argumenty: tablicę (dokładniej: wskaźnik do niej), indeks do elementu, gdzie ma być zapisana wartość i tę wartość. Najpierw funkcja sprawdza, czy otrzymany indeks nie jest większy od lub równy bieżącej liczbie elementów w tablicy. Ta liczba jest przechowywana w zmiennej lokalnej `length` zadeklarowanej jako `static`, a więc takiej, która nie jest niszczona po zakończeniu działania funkcji, ale istnieje przez cały czas wykonania programu. Jej wartością początkową jest 0. Jeśli warunek jest spełniony, to funkcja zwiększa liczbę elementów tablicy o wartość stałej `DELTA`, wynoszącą 10 i przydziela pamięć na tablicę przy pomocy `realloc()`. Wartość zwracana przez tę ostatnią funkcję zapisywana jest we wskaźniku `new_array`. Ponieważ przydział może się nie powieść, to funkcja `add_element()` najpierw sprawdza, czy wskaźnik `new_array` nie jest pusty, zanim przypisze jego wartość do wskaźnika `array`.

Zmienne dynamiczne

Przykład — dynamiczna tablica

Następnie funkcja `add_element()` dokonuje takiego samego sprawdzenia wskaźnika `array` (wcześniejsze przydziały też mogły się nie powieść) i jeśli nie jest on pusty, to zapisuje w elemencie tablicy o przekazanym indeksie otrzymaną wartość.
Proszę zwrócić uwagę, że możliwe jest zwrócenie adresu zmiennej z funkcji. Wymaga to jedynie, oprócz poprawnego użycia instrukcji `return`, zadeklarowania jako typu wartości zwracanej przez funkcję, określonego typu wskaźnikowego. Ściśle zakazanym jest zwracanie adresu automatycznej zmiennej lokalnej (w tym parametru), uzyskanego poprzez operator (&) lub innym sposobem. Taka zmienna przestaje istnieć po zakończeniu działania funkcji, zatem ten adres od razu staje się nieprzydatny. Jedynym wyjątkiem od tej reguły jest adres zmiennej lokalnej zadeklarowanej z użyciem słowa kluczowego `static`. Funkcja `add_element()` zwraca adres zmiennej (tablicy) dynamicznej, który tylko jest przechowywany w lokalnym wskaźniku. Oznacza to, że można go bezpiecznie zwrócić z funkcji.

Zmienne dynamiczne

Przykład — dynamiczna tablica

```
void print_array(const int *array,
                 const unsigned int number_of_values)
{
  if(array) {
    printf("W tablicy znajduje się %u wartości.\n",
                                        number_of_values);
    for (int i = 0; i < number_of_values; i++)
      printf("%d ", array[i]);
    puts("");
  }
}
```

Zmienne dynamiczne

Przykład — dynamiczna tablica

Funkcja `print_array()` wypisuje liczby z tablicy na ekranie. Jako argumenty przyjmuje tablicę i liczbę *wartości* w niej przechowywanych. Proszę zauważyć, że tablica może mieć więcej elementów, niż jest w niej przechowywanych wartości, bo za każdym razem kiedy jej wielkość jest zwiększana 10 nowych elementów jest do niej dodawanych. Warto również zwrócić uwagę na to, że zanim funkcja `print_array()` odwoła się do tablicy, najpierw sprawdza, czy wskaźnik na nią nie jest pusty. Może się zdarzyć tak, że ta tablica w ogóle nie zostanie utworzona przez funkcję `add_element()`, stąd taka weryfikacja jest konieczna. Oprócz wartości zgromadzonych w tablicy funkcja `print_array()` wyświetla także ich liczbę.

Zmienne dynamiczne

Przykład — dynamiczna tablica

```
int main(void)
{
    int *array = NULL;
    srand(time(0));
    unsigned int number_of_values = 1000 + rand()%1001;
    for(int i=0; i<number_of_values; i++)
        array = add_element(array,i, -5+rand()%26);
    print_array(array, number_of_values);
    free(array);
    return 0;
}
```

Zmienne dynamiczne

Przykład — dynamiczna tablica

W funkcji `main()` wskaźnik na tablicę jest inicjowany wartością `NULL`, co oznacza, że początkowo ta tablica nie istnieje. Program losowo wybiera liczbę wartości, które będą przechowywane w tej tablicy z przedziału $[1000, 2000]$. Sama tablica jest tworzona i wypełniana liczbami w pętli `for`. Za każdym razem, gdy funkcja `add_element()` wykryje, że nie ma już elementów w tablicy, aby zapisać nową liczbę, tworzy dodatkowych 10. Wartość ta została wybrana arbitralnie, jednakże im jest ona większa, tym więcej elementów może zostać nieużytych. Z drugiej strony im jest ona mniejsza tym częściej program musi przydzielać pamięć, a to jest czasochłonna operacja. Jeśli program miałby być używany produkcyjnie, to zapewne konieczne byłoby wybranie jakiejś innej liczby. Po wypełnieniu tablicy jej zawartość jest wypisywana na ekranie przez `print_array()`, a potem program zwalnia pamięć przydzieloną na tablicę przy użyciu funkcji `free()`.

Zmienne dynamiczne

Przykład — dynamiczna macierz

Podwójne wskaźniki umożliwiają tworzenie macierzy (dwuwymiarowych tablic) dynamicznych. Następny z prezentowanych na wykładzie programów pozwala użytkownikowi określić wymiary macierzy (liczbę wierszy i kolumn), a następnie tworzy ją w sposób dynamiczny i wypełnia pseudolosowymi liczbami całkowitymi z zakresu $[-10, 10]$.

Zmienne dynamiczne

Przykład — dynamiczna macierz

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

int **create(const int rows, const int columns)
{
    int **matrix = (int **)calloc(rows,sizeof(int *));
    if(matrix) {
        for(int i=0; i<rows; i++)
            matrix[i] = (int *)calloc(columns,sizeof(int));
    }
    return matrix;
}
```

Zmienne dynamiczne

Przykład — dynamiczna macierz

Funkcja `create()` jest odpowiedzialna za przydział pamięci dla macierzy. Najpierw tworzy ona tablicę wskaźników typu `int *`. Liczba elementów w tej tablicy jest określana przez wartość parametru `rows`. Jeśli ta operacja się powiedzie, to w pętli `for` przydzielana jest pamięć na tablice elementów typu `int`. Adresy tych tablic są zapisywane do elementów tablicy wskaźników. Liczba elementów w każdej z tablic liczb całkowitych jest określona wartością parametru `columns`. Na koniec funkcja zwraca adres tablicy wskaźników. Proszę zwrócić uwagę, że jest on początkowo przechowywany w podwójnym wskaźniku lokalnym o nazwie `matrix` oraz że typem wartości zwracanej przez funkcję też jest podwójny wskaźnik.

Zmienne dynamiczne

Przykład — dynamiczna macierz

```
void fill(int **matrix, const int rows, const int columns)
{
    for(int i=0; i<rows; i++)
        for(int j=0; j<columns; j++)
            matrix[i][j] = -10+rand()%21;
}

void print(int **matrix, const int rows, const int columns)
{
    for(int i=0; i<rows; i++) {
        for (int j = 0; j < columns; j++)
            printf("%4d", matrix[i][j]);
        puts("");
    }
}
```

Zmienne dynamiczne

Przykład — dynamiczna macierz

Funkcja `fill()` wypełnia macierz pseudolosowymi liczbami całkowitymi z przedziału $[-10, 10]$, a `print()` wypisuje jej zawartość na ekranie. Proszę zwrócić uwagę na podobieństwo ich kodu do kodu funkcji wykonujących podobne operacje, które definiowane były w zeszłym semestrze do obsługi „zwykłych" macierz. Te „zwykłe" macierze poprawnie nazywa się *macierzami, na które pamięć jest przydzielana statycznie* lub po prostu *macierzami statycznymi*. Różnica między opisywanymi funkcjami, a tymi z zeszłego semestru polega głównie na tym, że do tych pierwszych macierz jest przekazywana przy pomocy parametru będącego wskaźnikiem na wskaźnik oraz że liczba wierszy i kolumn też jest przekazywana przez parametry.

Zmienne dynamiczne

Przykład — dynamiczna macierz

```
void release(int **matrix, const int rows)
{
    for(int i=0; i<rows; i++)
        free(matrix[i]);
    free(matrix);
}
```

Zmienne dynamiczne

Przykład — dynamiczna macierz

Funkcja `release()` jest odpowiedzialna za zwolnienie pamięci przydzielonej dla macierzy. Najpierw zwalnia ona pamięć przydzieloną na poszczególne tablice elementów typu `int`, a następnie usuwa tablicę wskaźników.

Zmienne dynamiczne

Przykład — dynamiczna macierz

```
int main(void)
{
    srand(time(0));
    puts("Proszę podać liczbę wierszy i kolumn.");
    puts("Wiersze?");
    int rows = 0; scanf("%d",&rows);
    puts("Kolumny?");
    int columns=0; scanf("%d",&columns);
    int **matrix = create(rows,columns);
    if(matrix) {
        fill(matrix,rows,columns);
        print(matrix, rows, columns);
        release(matrix, rows);
    }
    return 0;
}
```

Zmienne dynamiczne

Przykład — dynamiczna macierz

W funkcji `main()` program pyta użytkownika o liczbę wierszy i kolumn, które powinna mieć macierz. Następnie jest ona tworzona za pomocą funkcji `create()`. Potem program sprawdza, czy tworzenie macierzy się powiodło i jeśli tak jest, to wypełnia ją liczbami pseudolosowymi przy użyciu funkcji `fill()`, wyświetla jej zawartość na ekranie wywołując `print()` i w końcu zwalnia przydzieloną na tę macierz pamięć za pomocą `release()`. Gdyby przydział pamięci przez `create()` się nie powiódł, to żadna z tych operacji nie zostałaby wykonana.
Program możne być udoskonalony poprzez dodanie w funkcji `create()` kodu, który sprawdzałby, czy pamięć na tablice liczb całkowitych została poprawnie przydzielona.

Zmienne dynamiczne

Podsumowanie

Wskaźniki i zmienne dynamiczne mogą posłużyć do tworzenia bardziej skomplikowanych struktur danych, niż zaprezentowane na tym wykładzie tablice. Kolejne z nich poznamy już wkrótce.

Jak czytać skomplikowane deklaracje?[1]

Przeglądając materiał dotyczący wskaźników na funkcje możemy się przekonać, że zmienne w języku C mogą mieć skomplikowane deklaracje. Wskaźniki na funkcje są tylko jednym z licznych przykładów. Powstaje zatem pytanie, w jaki sposób odczytać tak skomplikowany zapis, aby dowiedzieć się z jakiego typu zmienną mamy do czynienia w programie. Okazuje się, że jest na to stosunkowo prosty przepis:

Reguła

Zacznij od nazwy (lub najbardziej wewnętrznych nawisów okrągłych, jeśli żaden identyfikator nie jest obecny). Popatrz w prawo nie wychodząc poza prawy nawias okrągły. Powiedz co widzisz. Popatrz teraz w lewo, nie wychodząc poza lewy nawias okrągły. Powiedz co widzisz. Wyjdź poziom wyżej, poza bieżącą parę nawiasów okrągłych i powtórz to co zrobiłaś/zrobiłeś przed chwilą. Odczytywanie skończy się w momencie kiedy wypowiesz typ zmiennej lub typ wartości zwracanej przez funkcję.

[1] Na podstawie artykułu Terence'a Parra opublikowanego tutaj: https://parrt.cs.usfca.edu/doc/how-to-read-C-declarations.html

Jak czytać skomplikowane deklaracje?

Kolejne slajdy zawierają kilka przykładów deklaracji wraz z ich opisem. Nazwy użytych w przykładach zmiennych są celowo jednoliterowe, aby nie zdradzać czym są te zmienne.

Jak czytać skomplikowane deklaracje?

Przykład nr 1

Przykład

```
int *a[10];
```

Jak czytać skomplikowane deklaracje?

Przykład nr 1

Przykład

```
int *a[10];
```

Odpowiedź

Zmienna a jest tablicą 10 wskaźników typu `int`.

Jak czytać skomplikowane deklaracje?

Przykład nr 2

Przykład

```
int (*x) (int *, int *);
```

Jak czytać skomplikowane deklaracje?

Przykład nr 2

Przykład

```
int (*x) (int *, int *);
```

Odpowiedź

Zmienna `x` jest wskaźnikiem na funkcję, która ma dwa parametry będące wskaźnikami typ `int` i która zwraca wartość typu `int`.

Jak czytać skomplikowane deklaracje?

Przykład nr 3

Przykład

```
int (*(*v)[])();
```

Jak czytać skomplikowane deklaracje?

Przykład nr 3

Przykład

```
int (*(*v)[])();
```

Odpowiedź

Zmienna `v` jest wskaźnikiem na tablicę wskaźników na funkcje, które przyjmują nieokreśloną liczbę argumentów wywołania i zwracają wartość typu `int`.

Pytania

?

KONIEC

Dziękuję Państwu za uwagę!