Podstawy Programowania 2

Wskaźniki i zmienne dynamiczne

Arkadiusz Chrobot

Zakład Informatyki

24 lutego 2020

Plan

1. Wskaźniki - co już wiemy?
2. Wskaźniki na funkcje
3. Zmienne dynamiczne
4. Jak czytać skomplikowane deklaracje?

Wskaźniki - krótka powtórka

Zmienne wskaźnikowe, nazywane krótko *wskaźnikami*, nie przechowują bezpośrednio danych, ale adres pojedynczej komórki pamięci lub adres pierwszej komórki, z grupy przyległych komórek pamięci, które zawierają informacje przetwarzane przez program. Zmienna tego typu jest deklarowana według następującego wzorca:

```
typ_danych *nazwa_wskaźnika;
```

Typ danych określa w tym przypadku nie typ informacji przechowywanej bezpośrednio przez wskaźnik (tą informacją jest zawsze adres) lecz typ danych przechowywany w *zmiennej wskazywanej*, czyli wspomnianej wcześniej komórce lub grupie komórek. Zatem deklarację: `int *integer_pointer;` powinniśmy przeczytać jako: *wskaźnik na zmienną typu* `int`, ale zazwyczaj skracamy ten opis do *wskaźnik typu* `int`. Możliwe jest również zadeklarowanie wskaźnika, który wskazuje na dane nieokreślonego typu. Jego deklaracja wykonywana jest według następującego wzorca:

```
void *nazwa_wskaźnika;
```

Typ danym wskazywanym przez taki wskaźnik można nadać stosując rzutowanie.

Wskaźniki - krótka powtórka

Wartości wskaźników

Wskaźniki zawsze zawierają wyłącznie *adresy komórek pamięci*. Wskaźnik, który na nic nie wskazuje jest nazywany *wskaźnikiem pustym* i ma wartość określoną stałą `NULL`. Standard ISO C99 pozwala zamiast tej stałej użyć po prostu liczby `0`. Wskaźniki zadeklarowane jako zmienne globalne są domyślnie wskaźnikami pustymi. Wskaźniki lokalne, czyli zadeklarowane jako zmienne lokalne nie są domyślnie inicjowane i mają wartość przypadkową. W żargonie informatycznym określa się je mianem *dzikich wskaźników*. Ich użycie przed prawidłową inicjacją jest bardzo niebezpieczne, ponieważ można w ten sposób uszkodzić informacje w dowolnej komórce pamięci należącej do programu, co może prowadzić do jego błędnej pracy i zakończenia w trybie awaryjnym. Funkcje z rodziny `printf()` pozwalają wypisać na ekranie lub zapisać w pliku adres przechowywany w zmiennej wskaźnikowej za pomocą ciągu formatującego `%p`. Nie powinno się stosować tego ciągu z funkcjami z rodziny `scanf()`.

Wskaźniki - krótka powtórka

Dostęp do danych

Jeśli wskaźnik wskazuje na dane określonego typu, to dostęp do nich możemy uzyskać za pomocą *dereferencji* zmiennej wskaźnikowej. W języku C operatorem dereferencji jest * („gwiadzka” lub asterysk), czyli ten sam symbol, którego używamy do deklaracji wskaźników. Aby dokonać dereferencji wcześniej zadeklarowanego wskaźnika należy przed jego nazwą umieścić ten właśnie symbol. Inicjacji wskaźnika można dokonać przypisując mu adres zmiennej globalnej lub lokalnej. Adres takiej zmiennej można uzyskać za pomocą operatora & (ampersand), nazywanego operatorem „wyłuskania”. Wystarczy umieścić go przez nazwą zmiennej. Jeśli wskaźnik wskazuje na strukturę lub unię, to dostęp do jej pól można uzyskać na dwa sposoby:

```
nazwa_struktury->nazwa_pola
```

lub:

```
(*nazwa_struktury).nazwa_pola
```

Wskazane jest używanie pierwszego sposobu, gdyż jest krótszy i bardziej czytelny.

Wskaźniki - krótka powtórka

Arytmetyka wskaźników

Wskaźniki, a właściwie adresy w nich zawarte mogą być porównywane za pomocą tych samych operatorów, które używane są do porównywania np. liczb całkowitych. Dodatkowo język C umożliwia stosowanie tzw. *arytmetyki wskaźników*, tzn. do wskaźnika można dodać lub odjąć liczbę całkowitą. Wskaźniki można od siebie także odejmować, nie można ich natomiast dodawać. Ze wskaźnikami można stosować operatory inkrementacji i dekrementacji. Arytmetyka wskaźników może być wykorzystana np. w obsłudze tablic. Należy ją jednak stosować bardzo ostrożnie, a w przypadku kiedy można uniknąć jej użycia należy to zrobić.

Wskaźniki - krótka powtórka

Zastosowanie wskaźników w funkcjach

Wskaźniki mogą być użyte jako parametry funkcji. O argumentach przekazywanych przez takie parametry mówimy, że są przekazywane przez wskaźnik lub przez adres. Parametrami wskaźnikowymi wewnątrz funkcji posługujemy się jak zwykłymi wskaźnikami. Nazwy argumentów przekazywanych za pomocą tych wskaźników muszą być poprzedzone operatorem wyłuskania (&). Istnieją dwa wyjątki od tej reguły. Pierwszym są wskaźniki, a drugim tablice jednowymiarowe, których nazwy są traktowane w języku C jako wskaźniki. Typy argumentów muszą być takie same jak typy wskaźników pod które są one podstawiane. Tu również istnieje wyjątek w postaci wskaźnika bez określonego typu. Zmiana wartości wskazywanej przez parametr wskaźnikowy jest zmianą wartości argumentu podstawionego pod ten wskaźnik - jest on zmienną wskazywaną.
Funkcje mogą zwracać wartości wskaźników (adresy). Wystarczy, aby jako typ wartości przez nie zwracanych był podany typ wskaźnikowy. Nie wolno zwracać jako wartości takich funkcji adresów zmiennych lokalnych.

Wskaźniki - krótka powtórka

Przykład

```c
#include <stdio.h>

int main(void)
{
    int *pointer = NULL;
    int variable = 3;
    pointer = &variable;
    printf("Wartość wskazywanej zmiennej: %d\n",*pointer);
    printf("Adres przechowywany we wskaźniku: %p\n",pointer);
    variable++;
    printf("Wartość wskazywanej zmiennej: %d\n",*pointer);
    *pointer+=1;
    printf("Wartość wskazywanej zmiennej: %d\n",variable);
    return 0;
}
```

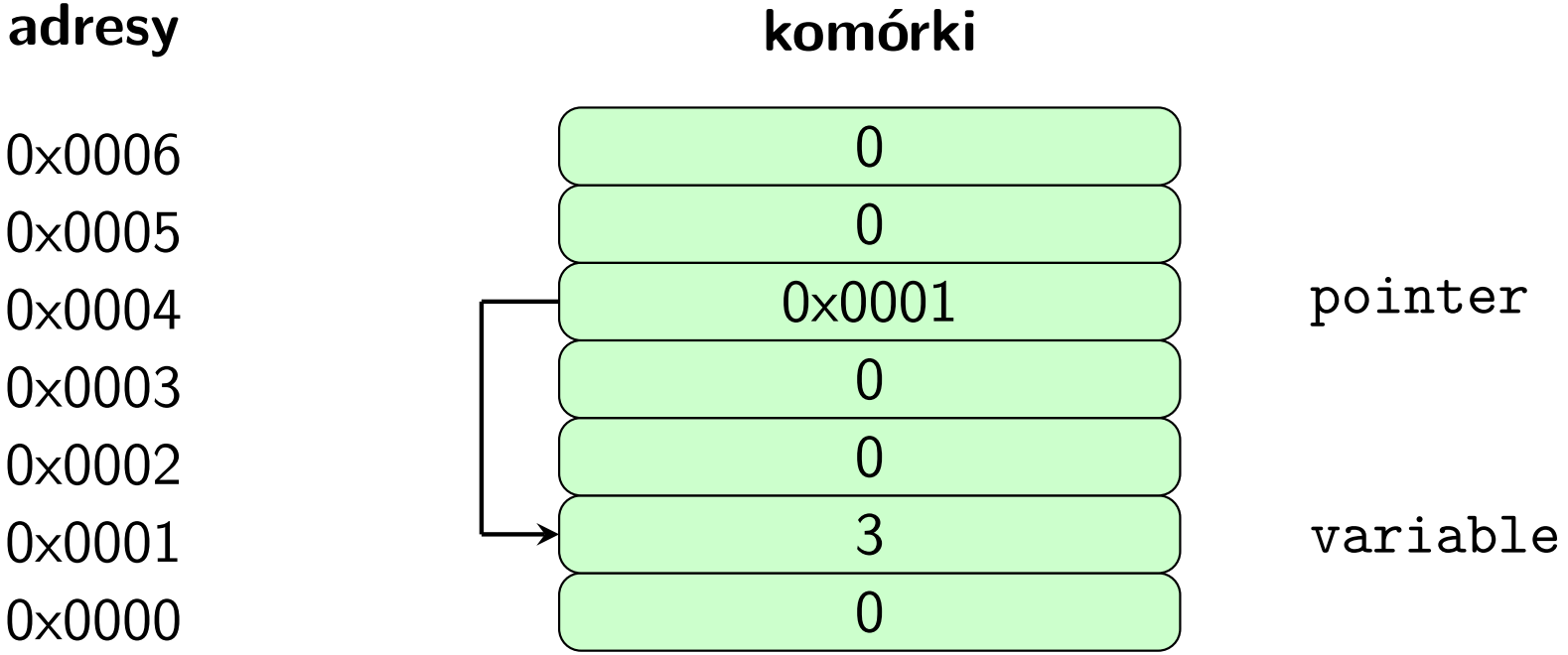
Wskaźniki - krótka powtórka

Komentarz do przykładu

Śledząc wykonanie przykładowego programu możemy przekonać się, że wartość zmiennej wskazywanej można zmienić za pomocą wskaźnika, jak również zmianę wartości tej zmiennej można za jego pomocą odczytać. Proszę zwrócić uwagę na różnicę między odczytaniem wartości wskaźnika (adresu, który przechowuje), a odczytem wartości zmiennej przez niego wskazywanej.

Wskaźniki - krótka powtórka

Aby lepiej zrozumieć jak działają wskaźniki, przedstawmy sobie bardzo uproszczony model pamięci, taki w którym każda zmienna ma wielkość jednej komórki. Tak mogłoby wyglądać rozmieszczenie zmiennych z przykładowego programu w tej pamięci:



Wskaźniki - krótka powtórka

Rysunek z poprzedniego slajdu ma charakter czysto poglądowy. Zaprezentowane na nim adresy komórek są sztucznie dobrane tak, aby w miarę prosto przedstawić obraz pamięci operacyjnej z podaną zawartością. W rzeczywistości rozmieszczenie zmiennych nie musi być takie samo, jak na rysunku. Wprowadzono na nim jeszcze jedno uproszczenie - przyjęto, że pojedyncza komórka pamięci jest na tyle pojemna, aby móc przechowywać i adres (czyli służyć jako wskaźnik) i wartość typu `int`. W rzeczywistości komórki pamięci zazwyczaj mają wielkość jednego bajta, a rozmiar wskaźników i zmiennych typu `int` jest zależny od budowy komputera, na którym program jest wykonywany. Obecnie najczęściej spotyka się rozmiar ośmiu (dla komputerów 64-bitowych) i czterech (dla komputerów 32-bitowych) bajtów. Oznacza to, że zarówno adres, jak i liczba typu `int` są zapisywane w więcej niż jednej komórce pamięci. Adresy zapisuje się, tak jak na rysunku, liczbami szesnastkowymi. Jedna cyfra w liczbie szesnastkowej odpowiada 4 bitom, a więc na rysunku adresy są 16-bitowe.

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("Hello there!");
}

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.

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 sobie, jakie główne rodzaje zmiennych znamy 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 zmienna 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 - tablica dynamiczna

Kolejne slajdy zawierają kod źródłowy przykładowego programu, który korzysta z dynamicznie utworzonej tablicy. Do przydzielenia pamięci na tę tablicę zostanie użyta funkcja `calloc()`.

Zmienne dynamiczne

Przykład - komentarz

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

#define NUMBER_OF_ELEMENTS 20

void fill_array(int *array)
{
    int i;
    for(i=0;i<NUMBER_OF_ELEMENTS;i++)
        array[i]=i;
}
```

Zmienne dynamiczne

Przykład - komentarz

Na poprzednim slajdzie umieszczony jest początek kodu źródłowego programu demonstrującego użycie tablicy dynamicznej. Jest on podobny do kodu programu posługującego się „zwykłą” tablicą. Zdefiniowano w nim stałą, która określa liczbę elementów, oraz funkcję, która wypełnia elementy tablicy wartościami ich indeksów. Parametr funkcji, którym jest wskaźnik, może być również zapisany jako `int array[]`, a więc dzięki temu, że nazwa tablicy jest traktowana w języku C jako wskaźnik, tablicę dynamiczną można obsługiwać za pomocą tej samej składni, co „zwykłą” tablicę.

Zmienne dynamiczne

Przykład - tablica dynamiczna

```
void print_array(int *array)
{
    int i;
    for(i=0;i<NUMBER_OF_ELEMENTS;i++)
        printf("%d ",array[i]);
    puts("");
}
```

Zmienne dynamiczne

Przykład - tablica dynamiczna

Poprzedni slajd zawiera funkcję, która wypisuje zawartość tablicy na ekran. Może ona być użyta także do wypisania wartości „zwykłej” tablicy.

Zmienne dynamiczne

Przykład - tablica dynamiczna

```
int main(void)
{
    int *array_pointer = (int *)calloc(NUMBER_OF_ELEMENTS,
                                       sizeof(int));
    if(array_pointer) {
        fill_array(array_pointer);
        print_array(array_pointer);
        free(array_pointer);
        array_pointer = NULL;
    }
    return 0;
}
```

Zmienne dynamiczne

Przykład - komentarz

W funkcji `main()` stworzona została tablica dynamiczna za pomocą wywołania funkcji `calloc()`. Jako pierwszy argument jej wywołania przekazano liczbę elementów tablicy, a jako drugi rozmiar pojedynczego elementu, czyli rozmiar typu `int`. Po sprawdzeniu, czy pamięć na tę tablicę została poprawnie przydzielona program wywołuje funkcje, które wypełniają jej elementy oraz wypisują je na ekran, a następnie tablica jest niszczona i wskaźnik na nią jest zerowany.

Zmienne dynamiczne

Przykład - symulacja `calloc()`

W następnym przykładzie wrócimy do poprzedniego programu i spróbujemy zastąpić funkcję `calloc()` własną funkcją o nazwie `allocate_array()`, która będzie działała tak samo jak ona.

Zmienne dynamiczne

Przykład - symulacja calloc()

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

#define NUMBER_OF_ELEMENTS 20

void *allocate_array(unsigned int number_of_elements,
                                    unsigned int element_size)
{
    unsigned long int array_size =
                            element_size * number_of_elements;
    void *array_pointer = malloc(array_size);
    if(array_pointer)
        array_pointer = memset(array_pointer,0,array_size);
    return array_pointer;
}
```

Zmienne dynamiczne

Przykład - komentarz

Do programu został włączony dodatkowy plik nagłówkowy o nazwie `string.h`, ponieważ będziemy w nim korzystać z funkcji `memset()`. Zdefiniowana w nim funkcja `allocate_array()` jest odpowiednikiem funkcji `calloc()`. Przez jej parametry przekazywana jest liczba elementów tworzonej tablicy i rozmiar pojedynczego elementu. Najpierw funkcja ta oblicza całkowitą wielkość tablicy poprzez pomnożenie wartości obu parametrów. Potem alokuje pamięć na tę tablicę za pomocą wywołania `malloc()`. Po upewnieniu się, że pamięć została przydzielona, tj. wskaźnik `array_pointer` ma wartość różną od zera lub `NULL`, zawartość utworzonej tablicy jest zerowana za pomocą wywołania funkcji `memeset()`. Niezależnie od tego czy tablica została utworzona i wyzerowana, czy też nie, zwracana jest przez funkcję wartość wskaźnika `array_pointer`. Jest to zgodne z zachowaniem (inaczej semantyką) funkcji `calloc()`. Proszę zauważyć, że ten wskaźnik, ma taki sam typ, jak wartość zwracana przez funkcję - `void *`. Warto także zauważyć, że wynik mnożenia z pierwszego wiersza ciała funkcji jest zapisywany w typie dwa razy większym od typów jego argumentów, celem uniknięcia przepełnienia.

Zmienne dynamiczne

Przykład - symulacja `calloc()`

```
void fill_array(int *array)
{
    int i;
    for(i=0;i<NUMBER_OF_ELEMENTS;i++)
        array[i]=i;
}

void print_array(int *array)
{
    int i;
    for(i=0;i<NUMBER_OF_ELEMENTS;i++)
        printf("%d ",array[i]);
    puts("");
}
```

Zmienne dynamiczne

Przykład - komentarz

Dwie funkcje zaprezentowane na poprzednim slajdzie są takie same jak w poprzednim programie.

Zmienne dynamiczne

Przykład - symulacja `calloc()`

```
int main(void)
{
    int *array_pointer = (int *)
            allocate_array(NUMBER_OF_ELEMENTS, sizeof(int));
    if(array_pointer) {
        fill_array(array_pointer);
        print_array(array_pointer);
        free(array_pointer);
        array_pointer = NULL;
    }
    return 0;
}
```

Zmienne dynamiczne

Przykład - komentarz

W funkcji `main()` programu wywołanie `calloc()` zostało zastąpione wywołaniem funkcji `allocate_array()`. Jak można się przekonać porównując oba programy, sposób korzystania z tych funkcji jest taki sami.

Zmienne dynamiczne

Przykład - tablica o zmiennym rozmiarze

Zaletą zmiennych dynamicznych jest to, że ich wielkość może być określona *w trakcie wykonania programu*. Spróbujemy teraz przerobić program tworzący tablicę, tak, aby użytkownik mógł określić jak wiele elementów będzie ona posiadała. Jej wielkość zatem będzie określoną wartością nie stałej, a zmiennej. Standard ISO C99 pozwala w ten sposób deklarować tablice lokalne, ale już w nowszym ISO C11 ten sposób ma być opcjonalny. Nie jest wykluczone, że w przyszłości będzie on w ogóle zabroniony. Lepszym pomysłem jest zatem wykorzystanie w tym celu zmiennych dynamicznych i następny przykład pokazuje jak to zrobić.

Zmienne dynamiczne

Przykład - tablica o zmiennym rozmiarze

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

void fill_array(int *array, unsigned int number_of_elements)
{
    int i;
    for(i=0;i<number_of_elements;i++)
        array[i]=i;
}

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

Zmienne dynamiczne

Przykład - komentarz

Z początku programu usunięta została stała `NUMBER_OF_ELEMENTS`. Zostanie ona zastąpiona w programie zmienną o takiej samej nazwie, pisanej małymi literami. Funkcje `fill_array()` i `print_array()` zyskały dodatkowy parametr, przez który przekazywana jest liczba elementów, które posiada tablica.

Zmienne dynamiczne

Przykład - tablica o zmiennym rozmiarze

```
int main(void)
{
    puts("Ile elementów ma zawierać tablica?");
    unsigned int number_of_elements = 0;
    scanf("%u",&number_of_elements);
    int *array_pointer =
                (int *)calloc(number_of_elements, sizeof(int));
    if(array_pointer) {
        fill_array(array_pointer,number_of_elements);
        print_array(array_pointer,number_of_elements);
        free(array_pointer);
        array_pointer = NULL;
    }
    return 0;
}
```

Zmienne dynamiczne

Przykład - komentarz

W funkcji `main()` została zadeklarowana zmienna typu `unsigned int` o wspomnianej wcześniej nazwie. Do niej zapisywana jest liczba elementów tablicy podana przez użytkownika i na jej podstawie przydzielana jest pamięć na tę tablicę. Uruchamiając program łatwo można się przekonać, że liczba wyświetlanych wartości elementów tablicy jest równa liczbie podanej przez użytkownika przy uruchamianiu programu.

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 w raz 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!