Podstawy Programowania 2

Stos i jego zastosowania

Arkadiusz Chrobot

Zakład Informatyki

3 marca 2020

Plan

Abstrakcyjne struktury danych

Dzięki omówionym na poprzednim wykładzie funkcjom zarządzającym obszarem pamięci programu, który nazywamy stertą możemy tworzyć zmienne dynamiczne. Na tym nie kończą się jednak możliwości, jakie dają te funkcje. Za ich pomocą możemy tworzyć całe struktury danych nazywane *strukturami dynamicznymi*. Mają one tę podstawową zaletę, ze ich rozmiar nie musi być znany w trakcie pisania programu. Są one tworzone, zgodnie ze swoją nazwą, dynamicznie podczas działania programu, a ich wielkość ograniczona jest tylko dostępnym miejscem na stercie. Większość dynamicznych struktur danych, to także *abstrakcyjne struktury danych*, czyli takie, które nie stanowią integralnej części języka programowania, ale mogą być utworzone na bazie dostarczanych przez niego elementów. Aby stworzyć abstrakcyjną strukturę danych najpierw musimy określić jej typ nazywany *abstrakcyjnym typem danych*. Taki typ nie tylko określa jakie dane będą przechowywane w strukturze, ale również *operacje*, które będzie można na tej strukturze przeprowadzić. Pierwszą taką strukturą, z jaką się zapoznamy jest *stos*.

Stos - wprowadzenie

Stos (ang. *stack*) w informatyce jest abstrakcyjną strukturą danych, w której dane są zarządzane zgodnie z zasadą *Ostatni Nadszedł Pierwszy Wychodzi* (ang. *Last In First Out* - LIFO). Może ona być zrealizowana na kilka różnych sposobów. Jednym z nich jest dynamiczna struktura danych, której elementy są ze sobą powiązane za pomocą wskaźników. Stos jest także szczególnym przypadkiem innych struktur, które nazywamy *kolejkami*, a te z kolei są szczególnym przypadkiem *list*. Wszystkie te struktury będziemy poznawać na kolejnych wykładach. Aby lepiej jednak zrozumieć czym jest stos przyjmiemy teraz, że lista to ciąg połączonych ze sobą elementów z początkiem i końcem. Stos jest listą, w której operacje dodawania i usuwania elementów wykonywane są wyłącznie na jej końcu. Zazwyczaj stos ilustruje się jako listę pionową, której koniec nazywa się *szczytem* lub *wierzchołkiem* (ang. *top*).

Stos - implementacja

Ponieważ stos jest abstrakcyjną strukturą danych, to musimy dla niej stworzyć abstrakcyjny typ danych, czyli określić jakie dane będą w niej przechowywane i zdefiniować operacje, które będą na niej wykonywane. To zadanie musimy wykonać korzystając z elementów, które są dostarczane przez język C. Jego pierwszą część zrealizujemy za pomocą struktury, a drugą część przy pomocy funkcji. Załóżmy dla uproszczenia, że nasz przykładowy stos będzie przechowywał w swoich elementach liczby typu `int`. W dalszej części wykładu zostanie zaprezentowany stos przechowujący ciągi znaków.

Stos - implementacja

Typ bazowy stosu

```
struct stack_node {
    int data;
    struct stack_node *next;
};
```

Stos - implementacja

Poprzedni slajd prezentuje definicję przykładowej struktury określającej typ pojedynczego elementy stosu, nazywany inaczej *typem bazowym stosu*. Jest on strukturą, która zawiera dwa pola. Pierwsze pole służy do przechowywania danych. W innych przypadkach takich pól może być więcej i mogą one mieć bardziej złożone typy danych. W przykładzie jest tylko jedno takie pole o nazwie `data` i typie `int`. Drugie pole jest polem wskaźnikowym. Spotykane są stosy, w których takich pól może być więcej, ale minimum wynosi jedno. Proszę jednak zwrócić uwagę na typ tego wskaźnika. Jest to wskaźnik na strukturę, w której jest on zawarty. Mamy więc od czynienia ze strukturą o *rekurencyjnej* budowie. Oznacza, to że ten wskaźnik może wskazywać na inną strukturę, której typ jest taki sam, jak tej, w której jest on zwarty. Innymi słowy, dzięki temu wskaźnikowi elementy stosu mogą się łączyć ze sobą.

Stos - implementacja

Kolejne elementy stosu będą powiązane ze sobą za pomocą umieszczonych w nich pól wskaźnikowych. Aby jednak móc wykonywać operacje na stosie zawsze musimy wiedzieć gdzie jest jego szczyt. Do zapamiętania jego adresu będziemy potrzebowali osobnej zmiennej wskaźnikowej, która może być lokalna lub globalna. W programach może mieć ona różną nazwę, ale ogólnie określa się ją mianem *wskaźnika stosu* lub, bardziej dokładnie *wskaźnika szczytu stosu*.
Mając elementy przechowujące dane musimy do definicji typu stosu dodać jeszcze operacje, które będą na nim wykonywane. Najbardziej podstawowymi operacjami dla stosu są dwie: dodanie nowego elementu do stosu, określane angielską nazwą *push* i zdjęcie lub usunięcie elementu ze stosu - *pop*. Obie te operacje wykonywane są na szczycie stosu. Na wykładzie zdefiniujemy jeszcze trzecią operację, która jest także często spotykana, ale nieobowiązkowa - odczytanie wartości elementu szczytowego stosu, czyli po angielsku *peek*.

Stos - implementacja

Funkcja `push()`

Operacja *push* została zaimplementowana w postaci funkcji `push()`. Określimy dwa warunki (niezmienniki), które musi spełniać ta funkcja, abyśmy mogli ją uznać za poprawną:

1. Przed wykonaniem funkcji wskaźnik stosu musi wskazywać element będący na szczycie stosu, albo być wskaźnikiem pustym - w tym ostatnim przypadku będziemy mieli do czynienia z *pustym stosem*, albo innymi słowy nieistniejącym.
2. Funkcja w wyniku swojego działania zwróci wskaźnik stosu, który będzie albo taki sam, jak jej został przekazany - będzie to oznaczało, że nie udało się utworzyć i dodać nowego elementu do stosu, albo będzie wskazywał na nowy element, który znajdzie się na szczycie stosu - w tym ostatnim przypadku stos zwiększy się o jeden element.

Stos - implementacja

Funkcja push()

```
1  struct stack_node *push(struct stack_node *top, int number)
2  {
3      struct stack_node *new_node = (struct stack_node *)
4                  malloc(sizeof(struct stack_node));
5      if(new_node!=NULL) {
6          new_node->data = number;
7          new_node->next = top;
8          top = new_node;
9      }
10     return top;
11 }
```

Uwaga! Numery wierszy nie są częścią kodu źródłowego. Są one wprowadzone aby ułatwić jego opis.

Stos - implementacja

Funkcja `push()`

Funkcja `push()` przyjmuje dwa argumenty wywołania, które są podstawiane pod widoczne w definicji funkcji parametry. Pierwszym z tych argumentów jest wskaźnik na stos, a drugim liczba, która ma być zapisana w nowym elemencie stosu. Pierwszą czynnością wykonywaną wewnątrz funkcji jest przydział pamięci na nowy element (wiersze 3 i 4). Przebieg dalszych działań zależy od jej powodzenia. Jeśli się ona nie powiedzie, to wskaźnik `node_new` będzie miał wartość `NULL`, a funkcja zakończy swoje działanie zwracając poprzednią wartość wskaźnika stosu (parametr `top`). W przeciwnym przypadku wskaźnik `new_node` będzie zawierał adres nowego elementu. W polu `data` tego elementu zostanie zapisana wartość liczby, którą będzie on przechowywał (wiersz 6). Następnie w wierszu 7 w polu `next` nowego elementu zostanie zapisany adres bieżącego wskaźnika stosu. W ten sposób nowy element zostanie połączony ze stosem i stanie się jego nowym szczytem i to na niego teraz powinien wskazywać wskaźnik `top`, dlatego w wierszu 8 jest do tego wskaźnika zapisywany adres nowego elementu.

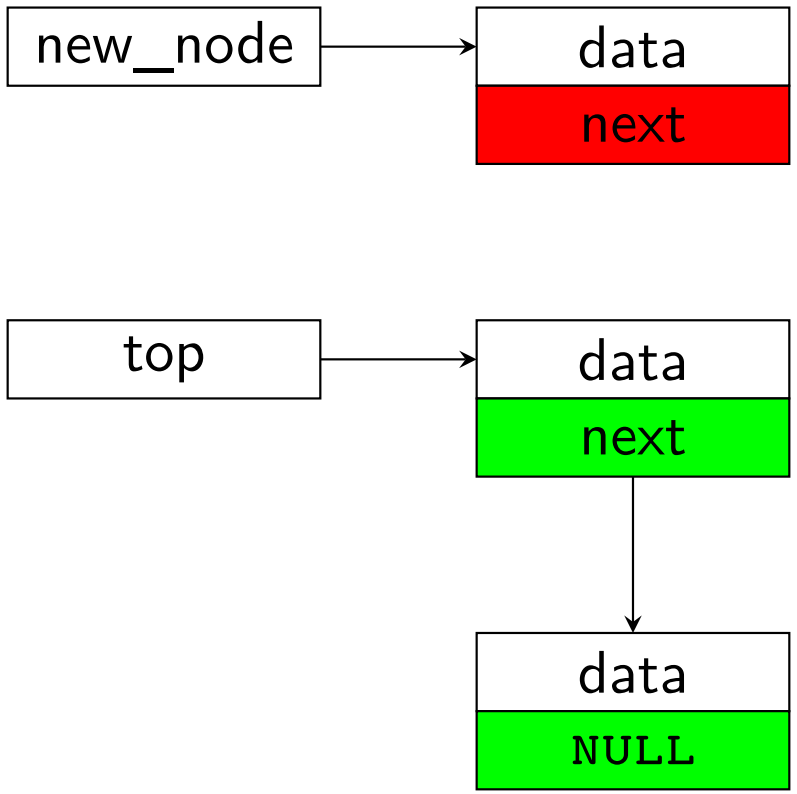
Stos - implementacja

Funkcja `push()`

Po wykonaniu tych czynności funkcja kończy działanie zwracając przy okazji wskaźnik stosu zawierający nowy adres.
Funkcję `push()` można zrealizować na kilka sposobów. Oprócz zaprezentowanego rozwiązania, stosuje się również takie, gdzie wskaźnik stosu jest przekazywany przez parametr będący podwójnym wskaźnikiem modyfikowanym wewnątrz funkcji. To rozwiązanie zostanie dokładniej opisane przy okazji objaśniania działania funkcji `pop()`.
Kolejne slajdy zawierają ilustrację udanego dodania nowego elementu do istniejącego stosu, składającego się z dwóch elementów. Proszę zwrócić uwagę, że pole `next` nowego elementu jest oznaczone początkowo na czerwono. Oznacza to, zgodnie z tym co zostało przedstawione na poprzednim wykładzie, że jest nieprawidłowym wskaźnikiem.

Stos - implementacja

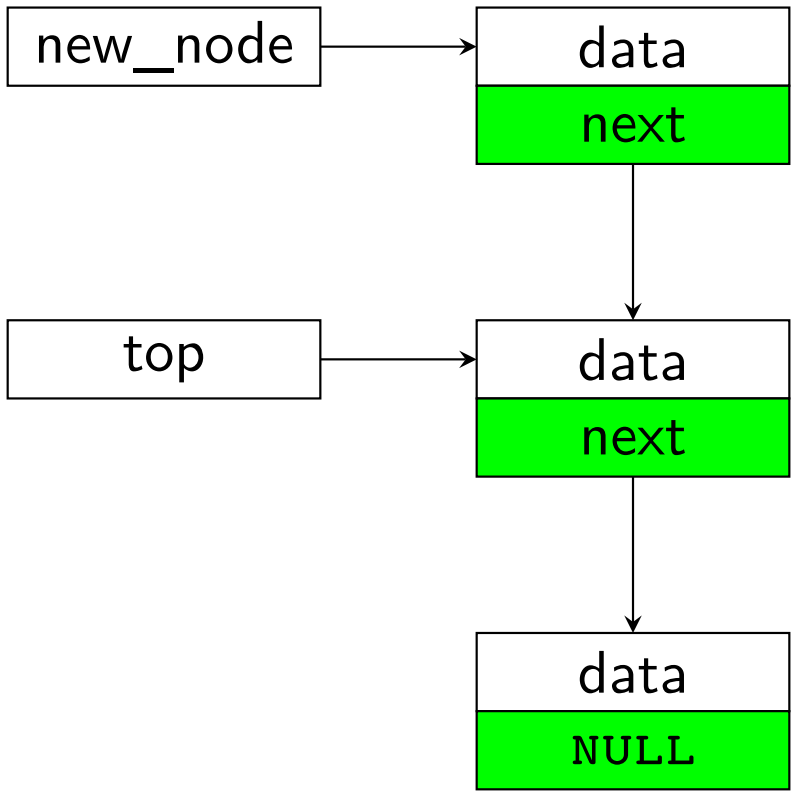
Funkcja `push()`



Przed wykonaniem wiersza nr 7 funkcji `push()`

Stos - implementacja

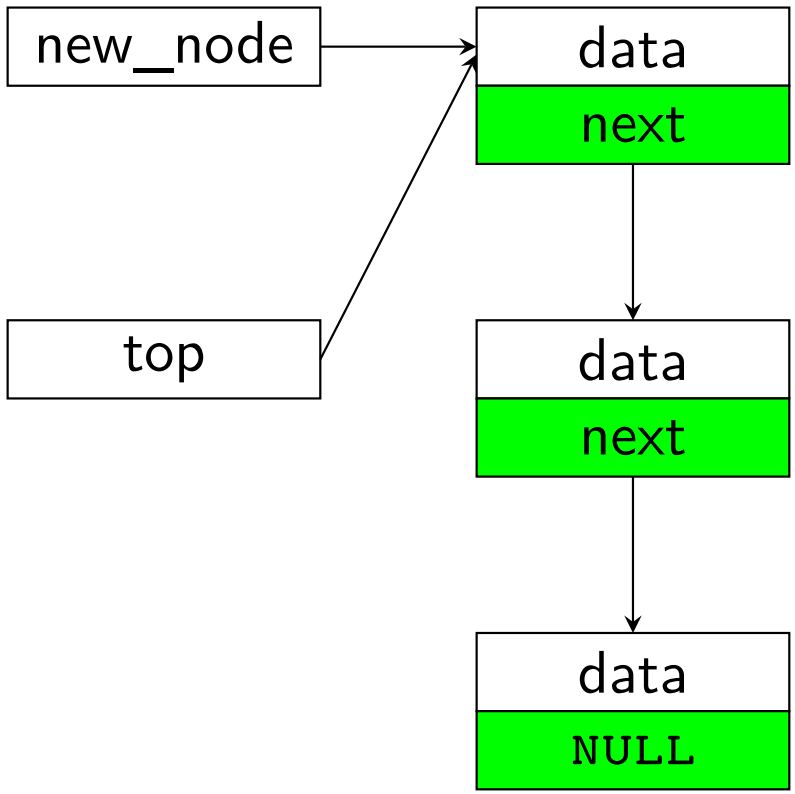
Funkcja `push()`



Przed wykonaniem wiersza nr 8 funkcji `push()`

Stos - implementacja

Funkcja `push()`



Przed wykonaniem wiersza nr 9 funkcji `push()`

Stos - implementacja

Funkcja `push()`

Proszę zwrócić uwagę, na charakterystyczną cechę pola `next` elementu znajdującego się na *dnie* stosu. Ma ono wartość `NULL`. Taka jego wartość oznacza, że mamy do czynienia z ostatnim elementem stosu, że jest to jego koniec i za nim nie ma już innych elementów tej struktury. Zastanówmy się, czy opisana wcześniej funkcja `push()` to gwarantuje. Okazuje się, że dzieje się tak tylko wtedy, gdy przy pierwszym jej wywołaniu związanym z tworzeniem pierwszego elementu danego stosu, zostanie jej przekazany wskaźnik stosu o wartości `NULL`. Wtedy jego wartość zostanie zapisana do pola `next` pierwszego i bieżąco jedynego, a wiec także ostatniego elementu stosu. Jeśli jednak funkcji zostanie przekazany nieprawidłowy wskaźnik, to jego wartość także zostanie umieszczona w tym polu i jest to bardzo niebezpieczna sytuacja, bo program nie będzie miał możliwości znalezienia końca stosu. Musimy zatem zadbać, aby przy *tworzeniu stosu* funkcji `push()` przekazać pusty wskaźnik, szczególnie wtedy, gdy ten wskaźnik jest zmienną lokalną.

Stos - implementacja

Funkcja `pop()`

Operacja *pop* zostanie zaimplementowana w postaci funkcji `pop()`. Podobnie jak dla funkcji `push()` wyznaczymy dla niej dwa niezmienniki, które muszą być spełnione, abyśmy mieli pewność, że została ona zrealizowana prawidłowo:

1. Przed wykonaniem funkcji wskaźnik stosu powinien wskazywać na istniejący stos lub być pusty.
2. Po wykonaniu funkcji wskaźnik stosu powinien wskazywać na istniejący stos, ale mniejszy o jeden element, lub powinien być pusty.

Stos - implementacja

Funkcja pop()

```
1  int pop(struct stack_node **top)
2  {
3      int result = -1;
4      if(*top) {
5          result = (*top)->data;
6          struct stack_node *tmp = (*top)->next;
7          free(*top);
8          *top = tmp;
9      }
10     return result;
11 }
```

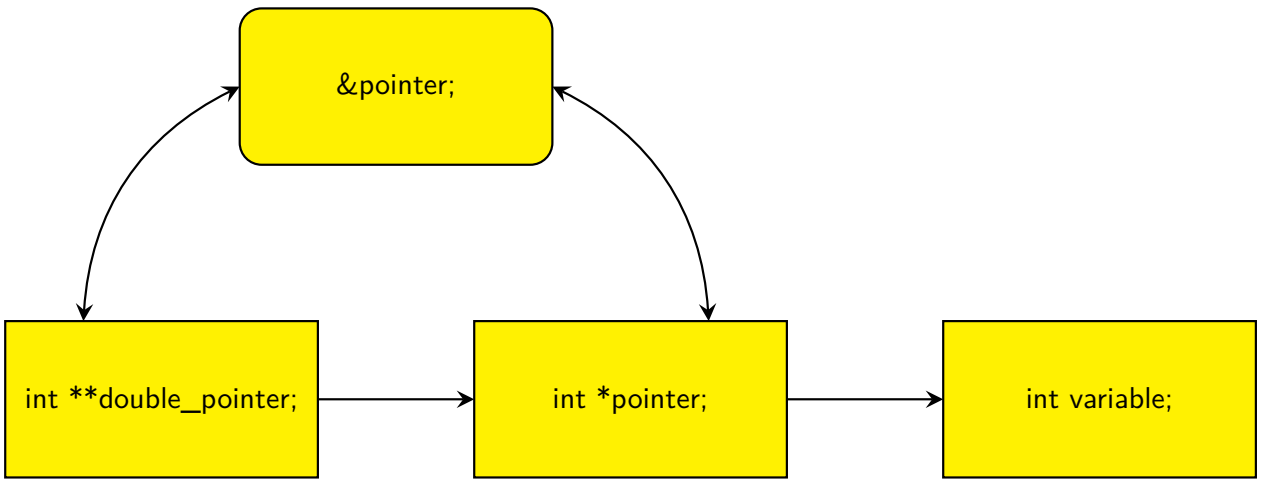
Stos - implementacja

Funkcja `pop()`

Funkcja `pop()` posiada jeden parametr, który jest podwójnym wskaźnikiem, albo inaczej *wskaźnikiem na wskaźnik*. Został on tutaj zastosowany ponieważ wewnątrz funkcji modyfikujemy stos, usuwając z jego szczytu jeden element. Oznacza to, że trzeba będzie zmodyfikować wartość wskaźnika stosu. Nie możemy postąpić tak jak w przypadku funkcji `push()`, bo będziemy tym razem chcieć, aby funkcja `pop()` zwróciła wartość usuwanego elementu, czyli to, co jest zapisane w jego polu `data`. Wyjściem jest tu właśnie podwójny wskaźnik, pod który podstawimy adres wskaźnika stosu. Jego działanie, dla ogólnego przypadku, ilustruje następny slajd.

Stos - implementacja

Funkcja pop()



Stos - implementacja

Funkcja `pop()`

Z zamieszczonej ilustracji wynika, że podwójny wskaźnik (`double_pointer`) może wskazywać na wskaźnik (`pointer`), który z kolei wskazuje na „zwykłą" zmienną (`variable`), choć równie dobrze może to być zmienna dynamiczna.
Aby odczytać wartość zmiennej `variable`, za pomocą podwójnego wskaźnika musimy zastosować dwa razy operator dereferencji (czyli `**double_pointer`). Jeśli zastosujemy go tylko raz (`*double_pointer`), to będziemy mogli odczytać zawartość wskaźnika `pointer`.

Stos - implementacja

Funkcja `pop()`

Funkcja `pop()` ma zmienną lokalną (`result`) typu `int`, która ma wartość początkową równą `-1`. Jeśli stos nie będzie istniał, to funkcja taką właśnie wartość umowną zwróci i zakończy swoje działanie. Istnienie stosu, czyli to, czy przekazany funkcji wskaźnik stosu nie jest pusty, sprawdzane jest na początku jej działania. Warunek `*top` jest skróconym zapisem warunku `*top!=NULL`. Jeśli jest on prawdziwy, to funkcja zapisuje do zmiennej `result` wartość bieżącego elementu szczytowego stosu (wiersz 5), a następnie zapamiętuje w zmiennej wskaźnikowej `tmp` adres następnego elementu stosu, który jest przechowywany w polu `next` elementu szczytowego (wiersz 6). Potem element szczytowy jest usuwany (wiersz 7). Tym samym wskaźnik stosu przestaje mieć prawidłową wartość - nie wskazuje na szczyt stosu. Aby naprawić tę sytuację w wierszu 8 zapisywany jest do niego adres przechowywany w zmiennej `tmp`. Po tym funkcja zwraca liczbę zapamiętaną w zmiennej `result` i kończy swoje działanie.

Stos - implementacja

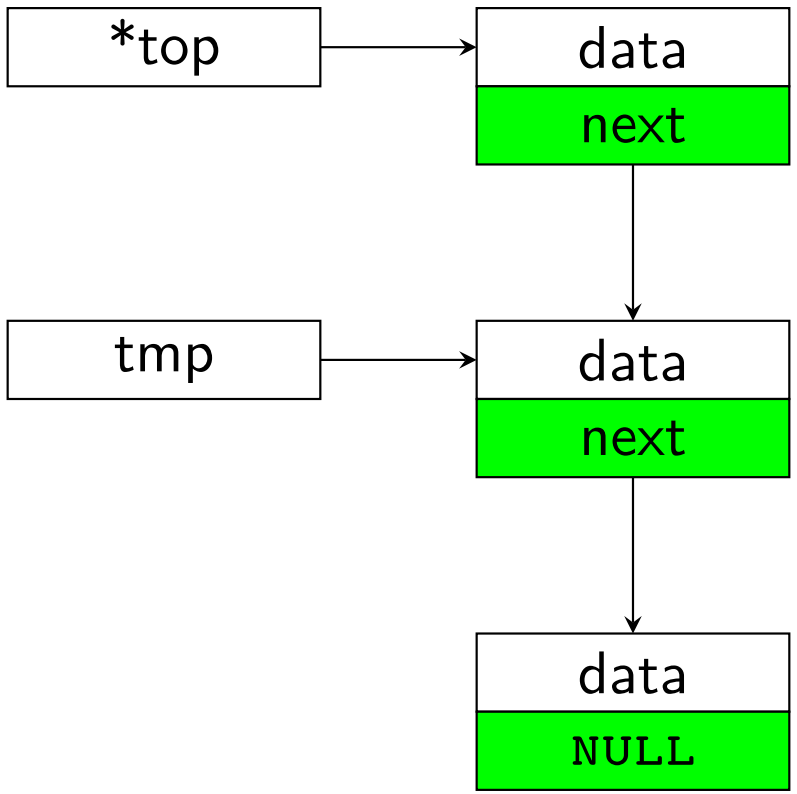
Funkcja `pop()`

Proszę zwrócić uwagę, że funkcja `pop()` usuwa poprawnie również ostatni element stosu. Jego wskaźnik `next` ma wartość `NULL` i taką też wartość uzyska wskaźnik `tmp`, po wykonaniu dla jednoelementowego stosu wiersza 6 funkcji. Po wykonaniu wiersza 8 ta wartość będzie nadana również wskaźnikowi stosu. Jest to oczekiwany wynik, gdyż w rezultacie usunięcia pojedynczego elementu ze stosu jednoelementowego powinniśmy uzyskać pusty stos.

Kolejne slajdy ilustrują działanie wybranych wierszy funkcji `pop()` w przypadku, gdy usuwa ona element ze stosu składającego się początkowo z trzech takich elementów. Proszę zwrócić uwagę, że ten rysunek jest uproszczony - po wywołaniu funkcji `free()` pamięć przeznaczona na element stosu nie znika, ani wskaźnik na niego wskazujący nie jest zerowany. Mimo to, nie powinniśmy się już odwoływać do tego elementu, ani nie powinniśmy posługiwać się tym wskaźnikiem do chwili ponownego nadania mu wartości, z powodów, które były objaśnione na poprzednim wykładzie.

Stos - implementacja

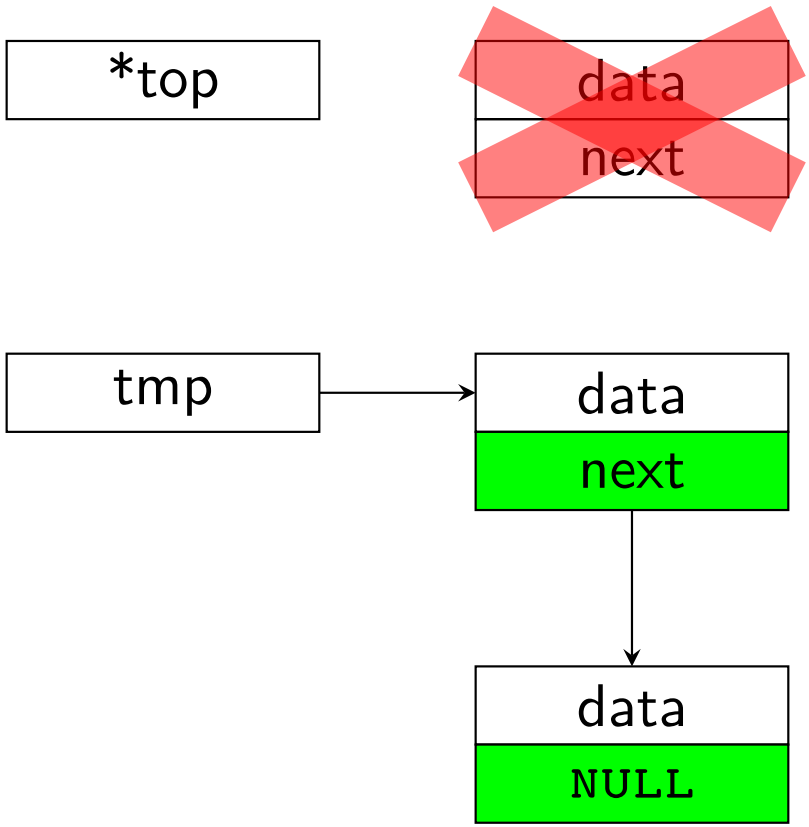
Funkcja pop()



Po wykonaniu wiersza nr 6 funkcji pop()

Stos - implementacja

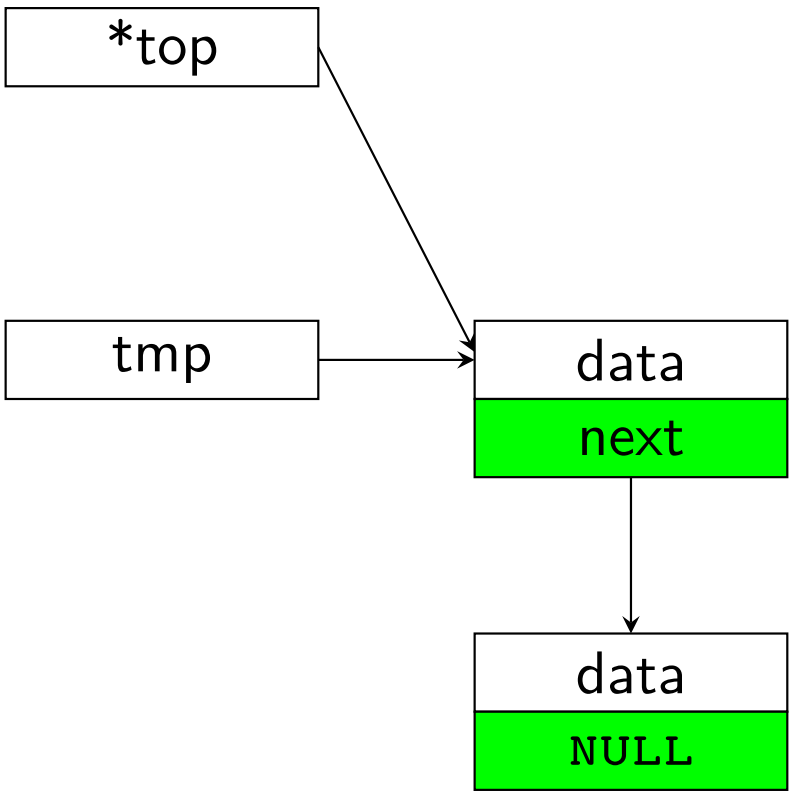
Funkcja pop()



Po wykonaniem wiersza nr 7 funkcji pop()

Stos - implementacja

Funkcja pop()



Po wykonaniu wiersza nr 8 funkcji pop()

Stos - implementacja

Funkcja peek() - opcjonalna

```
1  int peek(struct stack_node *top)
2  {
3      if(top)
4          return top->data;
5      else {
6          fprintf(stderr,"Stos nie istnieje.\n");
7          return -1;
8      }
9  }
```

Stos - implementacja

Funkcja `peek()` - opcjonalna

Operacja *peek* jest opcjonalna - nie trzeba jej oprogramowywać w każdej implementacji stosu. Na tym wykładzie zostanie jednak przedstawiona jej przykładowa realizacja w postaci funkcji `peek()`. Jej definicja jest stosunkowo prosta. Przez parametr przyjmuje ona wskaźnik na stos. Jeśli nie jest on pusty (warunek `top` jest skróconym zapisem warunku `top!=NULL`), to zwraca ona wartość elementu szczytowego tego stosu, a jeśli nie, to umieszcza komunikat w standardowym strumieniu diagnostycznym (najczęściej jest to ekran monitora) i zwraca taką samą wartość umowną jak funkcja `pop()`.

Stos - implementacja

Przykładowy program

Mamy zdefiniowane wszystkie elementy niezbędne do tego, aby stworzyć stos. Na kolejnych slajdach zostanie zaprezentowany prosty program, który korzysta z takiej struktury. Będzie on zapisywał na stosie liczby naturalne, a następnie je odczytywał i wypisywał na ekran. Oprócz włączonych plików nagłówkowych, które były opisywane już na poprzednich wykładach, program zawiera przedstawione na wcześniejszych slajdach funkcje oraz funkcję `main()`. Tylko ta funkcja będzie wymagała komentarza, który zostanie zamieszczony po slajdzie ją zawierającym.

Stos - implementacja

Przykładowy program

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

struct stack_node {
    int data;
    struct stack_node *next;
};
```

Stos - implementacja

Przykładowy program

```
struct stack_node *push(struct stack_node *top, int number)
{
    struct stack_node *new_node = (struct stack_node *)
                          malloc(sizeof(struct stack_node));
    if(new_node!=NULL) {
        new_node->data = number;
        new_node->next = top;
        top = new_node;
    }
    return top;
}
```

Stos - implementacja

Przykładowy program

```
int pop(struct stack_node **top)
{
    int result = -1;
    if(*top) {
        result = (*top)->data;
        struct stack_node *tmp = (*top)->next;
        free(*top);
        *top = tmp;
    }
    return result;
}
```

Stos - implementacja

Przykładowy program

```
int peek(struct stack_node *top)
{
    if(top)
        return top->data;
    else {
        fprintf(stderr,"Stos nie istnieje.\n");
        return -1;
    }
}
```

Stos - implementacja

Przykładowy program

```
int main(void)
{
    struct stack_node *top = NULL;
    srand(time(0));
    int i;
    for(i=1; i<6+rand()%5; i++)
        top=push(top,i);
    printf("Wartość ze szczytu stosu: %d\n",peek(top));
    while(top)
        printf("%d ",pop(&top));
    puts("");
    return 0;
}
```

Stos - implementacja

Przykładowy program

Wskaźnik na stos jest deklarowany jako zmienna lokalna funkcji `main()` o nazwie `top`. Początkowo stos ten jest pusty. W pętli `for` dodawane są do niego, przy pomocy wywołań funkcji `push()`, kolejne elementy, których wartości nadawane są przez licznik tej pętli (zmienna `i`). Wartością początkową tego licznika jest zawsze `1`, a wartość końcowa jest losowana z zakresu od 6 do `10`. W związku z tym przed uruchomieniem programu nie można z góry określić, ile liczb naturalnych znajdzie się na stosie. Po zakończeniu pętli program wypisuje na ekran wartość liczby znajdującej się w szczytowym elemencie stosu. Ta liczba jest uzyskiwana dzięki wywołaniu funkcji `peek()`. Następnie w pętli `while` elementy są zdejmowane ze stosu, aż do jego opróżnienia, a ich wartości są wypisywane na ekran. Warunek w tej pętli jest równoważny warunkowi `top!=NULL`. Wynik działania tego programu ujawnia bardzo ważną cechę stosu: *elementy są odczytywane ze stosu w odwrotnej kolejności do tej w jakiej zostały zapisane*.

Wycieki pamięci

Implementacja dynamicznej struktury danych może być obarczona wieloma poważnymi błędami. W przypadku stosu i pokrewnych struktur jednym z takich błędów jest brak spójności takiej struktury, polegający na tym, że jej elementy nie są prawidłowo ze sobą powiązane. Wyobraźmy np. sobie, że pewien nadgorliwy programista mógłby na początku funkcji `push()` zerować parametr `top`. To spowodowałoby, że każdy stworzony element stosu nie byłby powiązany z pozostałymi. Co gorsza, na żaden z tych elementów, poza ostatnim, nie wskazywałby żaden wskaźnik. Takich elementów nie można by było już usunąć ze sterty. Obszary tej pamięci przydzielone na nie byłby bezpowrotnie stracone do końca działania programu. Taki błąd nazywa się w żargonie informatycznym *wyciekiem* lub *wyciekami pamięci* (ang. *memory leaks*). W najgorszym przypadku może on prowadzić do wyczerpania miejsca na stercie. Pierwszą obroną przed nim jest jego unikanie, czyli dokładne przeanalizowanie implementacji wszelkich operacji na strukturze danych. Istnieją też narzędzia programowe, od programów debugujących, po specjalne biblioteki, które ułatwiają wykrywanie tego typu błędów. Niestety, nie są one częścią standardu języka C, gdyż ich działanie zależy od użytego komputera i systemu operacyjnego.

Stos - zastosowania

Program konwertujący z kodu NKB do dziesiętnego

Stos jest dosyć powszechnie używaną strukturą danych. Jako drugi przykład jego zastosowania zostanie podany program, który konwertuje zapis binarny liczby (konkretnie NKB) na zapis dziesiętny. Użycie do tego celu stosu zaimplementowanego w postaci struktury dynamicznej jest dosyć nieefektywnym pomysłem, ale chodzi w tym przykładzie o pokazanie i wykorzystanie zasady działania stosu. Na przyszłym wykładzie podobny pomysł zostanie lepiej zrealizowany. W programie zostanie wykorzystana funkcja `getch()` z biblioteki *curses*. Dzięki temu liczba binarna będzie odczytywana bit po bicie z klawiatur. Każdy z nich będzie zapisywany na stosie, w związku z tym najstarszy bit liczby trafi na dno stosu, a najmłodszy będzie na jego szczycie. Użycie biblioteki *curses* wymagało włączenia plików nagłówkowych `curses.h` i `locale.h`.

Stos - zastosowania

Program konwertujący z kodu NKB do dziesiętnego

```
#include<stdlib.h>
#include<curses.h>
#include<locale.h>

struct stack_node {
    int data;
    struct stack_node *next;
};
```

Stos - zastosowania

Konwersja z kodu NKB do dziesiętnego

```
struct stack_node *push(struct stack_node *top, int number)
{
    struct stack_node *new_node = (struct stack_node *)
                          malloc(sizeof(struct stack_node));
    if(new_node!=NULL) {
        new_node->data = number;
        new_node->next = top;
        top = new_node;
    }
    return top;
}
```

Stos - zastosowania

Konwersja z kodu NKB do dziesiętnego

```
int pop(struct stack_node **top)
{
    int result = -1;
    if(*top) {
        result = (*top)->data;
        struct stack_node *tmp = (*top)->next;
        free(*top);
        *top = tmp;
    }
    return result;
}
```

Stos - zastosowania

Konwersja z kodu NKB do dziesiętnego

```
void put_binary_on_stack(struct stack_node **top)
{
    int input = 0;
    do {
        input = getch();
        if(input=='0'||input=='1')
            *top=push(*top,input-'0');
    } while(input=='0'||input=='1');
}
```

Stos - zastosowania

Konwersja z kodu NKB do dziesiętnego

Na poprzednich slajdach zostały umieszczone kody źródłowe funkcji `push()` i `pop()`, które były już prezentowane wcześniej, stąd nie są one komentowane. Ostatni slajd zawiera kod funkcji, która odczytuje z klawiatury kolejne znaki podawane przez użytkownika i jeśli są to cyfry `0` lub `1`, to zapisuje je na stosie. Ta czynność trwa tak długo, jak długo użytkownik nie wciśnie klawisza reprezentującego inny znak niż podane cyfry. Funkcja ta ma jeden parametr, który jest podwójnym wskaźnikiem. Przez ten parametr będzie przekazywany wskaźnik na stos (początkowo pusty), który będzie w niej ulegał modyfikacji. Odczyt znaków z klawiatury odbywa się w pętli `do...while`, która kończy się po podaniu innego znaku niż możliwe wartości pojedynczego bitu. Każdy bit zapisywany jest w osobnym elemencie na stosie. Proszę zwrócić uwagę na wywołanie funkcji `push()`. Wartość przez nią zwracana jest zapisywana w zdereferencjonowanym wskaźniku `top`, który również w tej postaci jest przekazywany jej jako pierwszy argument. Jako drugi argument jest przekazywane wyrażenie `input-'0'`.

Stos - zastosowania

Konwersja z kodu NKB do dziesiętnego

Ponieważ cyfra bitu jest zapisana w zmiennej `input` jako kod ASCII znaku, to żeby otrzymać jej wartość liczbową musimy od niej odjąć kod ASCII znaku 0.

Stos - zastosowania

Konwersja z kodu NKB do dziesiętnego

```
int convert_binary_to_decimal(struct stack_node **top)
{

    int result = 0, base = 1;
    while(*top) {
        int digit = pop(top);
        result += digit * base;
        base *= 2;
    }
    return result;
}
```

Stos - zastosowania

Konwersja z kodu NKB do dziesiętnego

Właściwa konwersja jest dokonywana przez funkcję, której kod źródłowy został zaprezentowany na poprzednim slajdzie. Ponieważ wskaźnik stosu będzie modyfikowany w jej wnętrzu poprzez wywołanie funkcji `pop()`, to jest on do niej przekazywany przez parametr będący podwójnym wskaźnikiem. Wynik konwersji jest zapisywany w zmiennej lokalnej `result`. Każdy bit ze stosu, począwszy od najmłodszego, musi być pomnożony przez wagę (zmienna `base`), będącą odpowiednią potęgą dwójki ($2^0 = 1, 2^1 = 2, 2^2 = 4$, itd.). Stąd zmienna `base` po każdej iteracji pętli `while` jest mnożona przez liczbę 2. Wiersz wcześniej jest ona użyta do pomnożenia przez nią wartości bitu jeszcze wcześniej zdjętego ze stosu. Uzyskany iloczyn jest dodawany do sumy takich iloczynów obliczonych dla wcześniejszych (młodszych) bitów. Pętla kończy się w momencie opróżnienia stosu. W takiej sytuacji funkcja zwraca wartość zmiennej `result` i kończy swoje działanie.

Stos - zastosowania

Konwersja z kodu NKB do dziesiętnego

```
int main(void)
{
    if(setlocale(LC_ALL,"")==NULL) {
        fprintf(stderr,"Błąd inicjacji ustawień językowych!\n");
        return -1;
    }
    if(!initscr()) {
        fprintf(stderr,"Błąd inicjacji biblioteki curses!\n");
        return -1;
    }
```

Stos - zastosowania

Konwersja z kodu NKB do dziesiętnego

Poprzedni slajd prezentuje fragment początkowy funkcji `main()` programu, który zawiera kod inicjujący pracę biblioteki *curses* oraz ustawienia językowe.

Stos - zastosowania

Konwersja z kodu NKB do dziesiętnego

```
    printw("Proszę wprowadzić liczbę binarną i zakończyć ją\
dowolnym znakiem:\n");
    (void)refresh();
    struct stack_node *top = NULL;
    put_binary_on_stack(&top);
```

Stos - zastosowania

Konwersja z kodu NKB do dziesiętnego

W zaprezentowanym na poprzednim slajdzie fragmencie funkcji `main()` wypisywany jest na ekran komunikat instruujący użytkownika co ma zrobić i tworzony jest stos, na którym zapisywane są poszczególne bity wprowadzanej przez niego liczby binarnej, dzięki wywołaniu funkcji `put_binarny_on_stack()`.

Stos - zastosowania

Konwersja z kodu NKB do dziesiętnego

```
    printw("Ta liczba zapisana w kodzie dziesiętnym to: %d.\n",
                                        convert_binary_to_decimal(&top));
    (void)refresh();
    getch();
    if(endwin()==ERR) {
        fprintf(stderr,"Błąd funkcji endwin()!\n");
        return -1;
    }
    return 0;
}
```

Stos - zastosowania

Konwersja z kodu NKB do dziesiętnego

W ostatnim zaprezentowanym fragmencie kodu funkcji `main()` dokonywana jest konwersja umieszczonego na stosie zapisu binarnego liczby na zapis dziesiętny dzięki wywołaniu funkcji `convert_binary_to_decimal()`. Wyliczona przez nią wartość jest wypisywana na ekranie, a następnie program czeka, aż użytkownik naciśnie dowolny klawisz. Po zakończeniu tego oczekiwania wyłączane jest działanie biblioteki *curses* i program się kończy. Proszę zauważyć, że dzięki zastosowaniu stosu, zaimplementowanego jako dynamiczna struktura danych program może konwertować nawet stosunkowo długie liczby binarne. Ograniczeniami są tu tylko maksymalna wartość zmiennych typ `int`, do których zapisywany jest wynik konwersji i rozmiar sterty.

Stos - zastosowania

Obliczanie wyrażeń w ONP

Stos używany jest do wyznaczania wartości wyrażeń arytmetycznych zapisanych w Odwrotnej Notacji Polskiej (ONP) (ang. *Reverse Polish Notation*), nazywanej też notacją przyrostkową (ang. *postfix*). Notację tę opracował (wraz z innymi osobami) australijski informatyk i filozof Charles Hamblin, bazując na Notacji Polskiej (ang. *Polish Notation*), nazywanej też przedrostkową (ang. *prefix*), wymyślonej przez polskiego logika Jana Łukasiewicza. Obie notacje są beznawiasowe, tzn. nie wymagają nawiasów, aby określić kolejność działań w wyrażeniu arytmetycznym. W Notacji Polskiej wszystkie operatory dwuargumentowe poprzedzają swoje argumenty. W OPN wszystkie operatory dwuargumentowe znajdują się za swoimi argumentami. Następny slajd zawiera kilka przykładów wyrażeń arytmetycznych zapisanych w tradycyjnej notacji wrostkowej (ang. *infix*) oraz ONP.

Stos - zastosowania

Obliczanie wyrażeń w ONP

$2 + 2 \Rightarrow 2\ 2\ +$

$(5 - 2) * (4 + 1) \Rightarrow 5\ 2\ -\ 4\ 1\ +\ *$

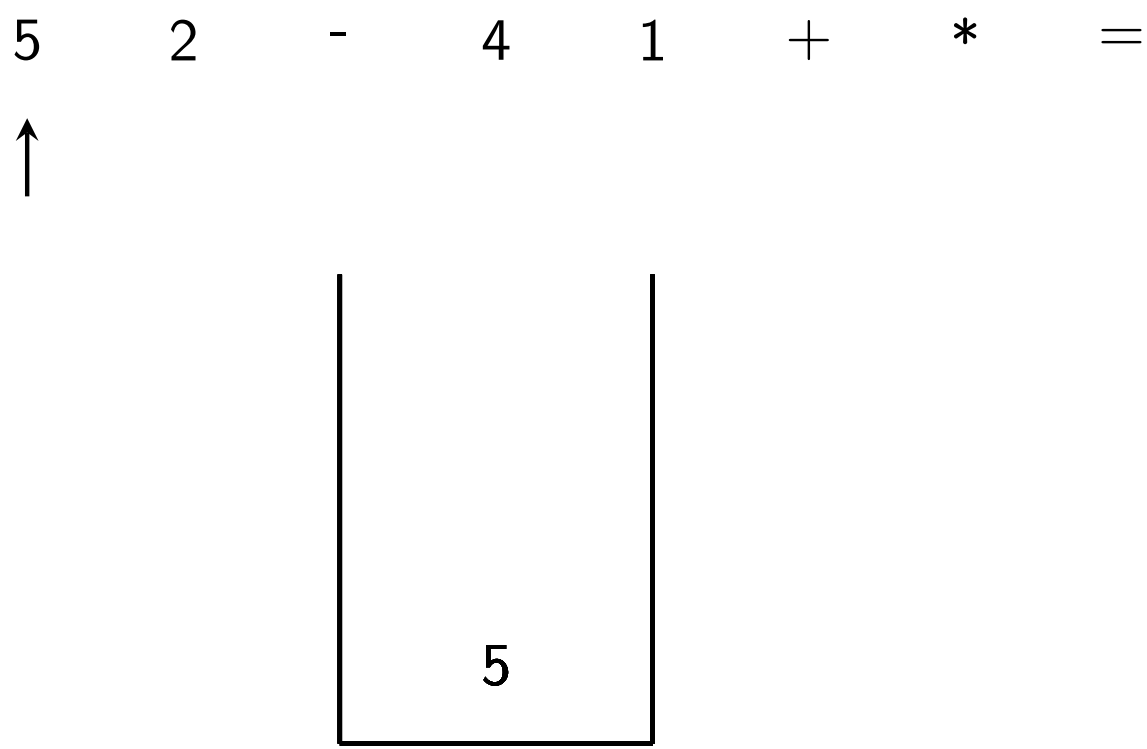
$(3 + 2) * 7 \Rightarrow 3\ 2\ +\ 7\ *$

$3 + 2 * 7 \Rightarrow 2\ 7\ *\ 3\ +$

Stos - zastosowania

Obliczanie wyrażeń w ONP

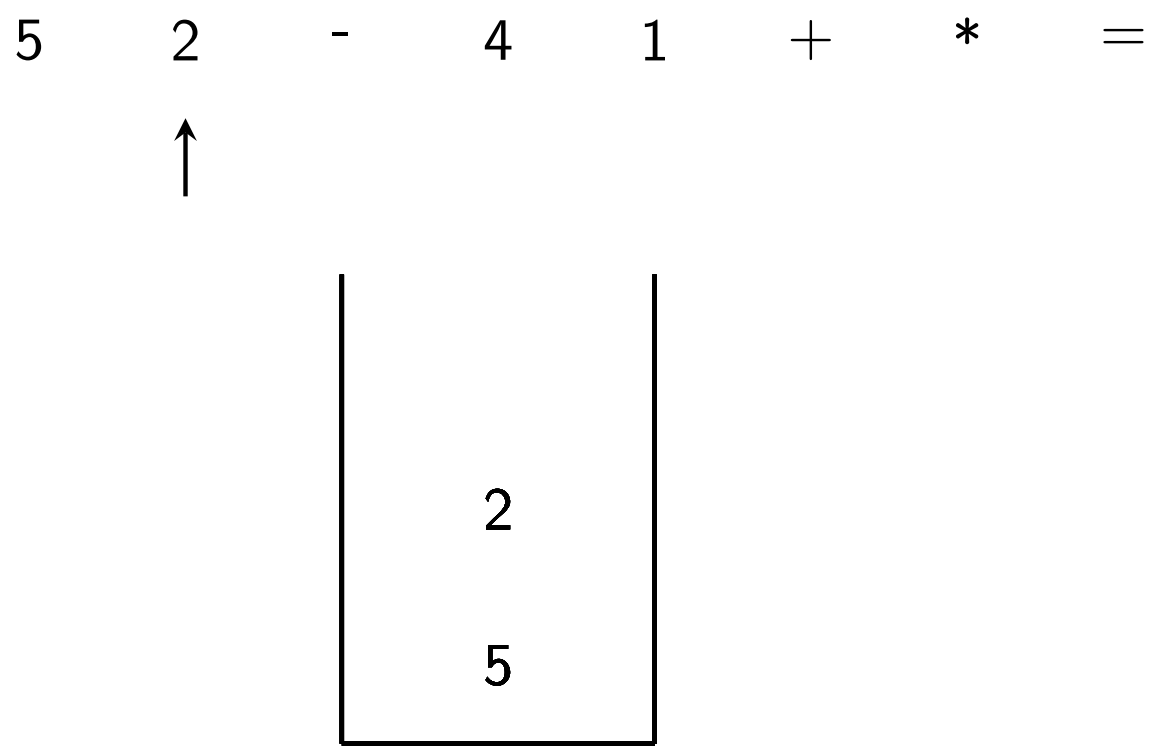
Zamieszczona poniżej animacja pokazuje związki między wyrażeniami w ONP, a stosem i obrazuje jak wyliczyć wartość takiego wyrażenia z jego użyciem.



Stos - zastosowania

Obliczanie wyrażeń w ONP

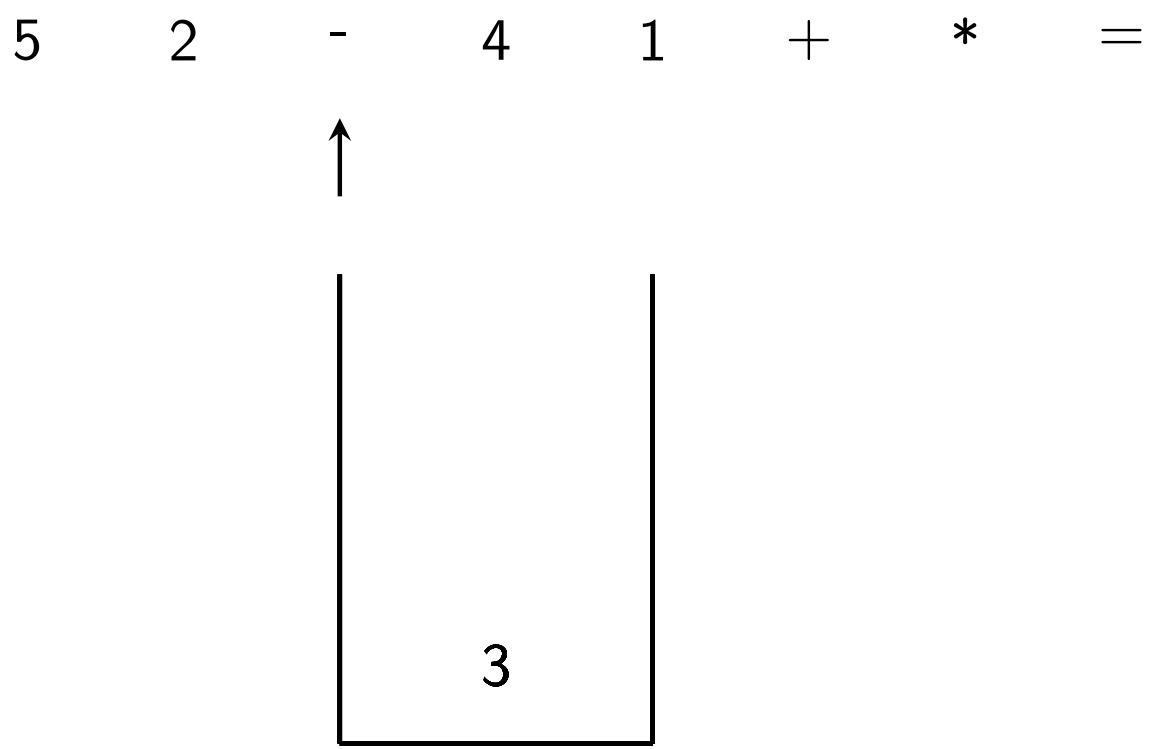
Zamieszczona poniżej animacja pokazuje związki między wyrażeniami w ONP, a stosem i obrazuje jak wyliczyć wartość takiego wyrażenia z jego użyciem.



Stos - zastosowania

Obliczanie wyrażeń w ONP

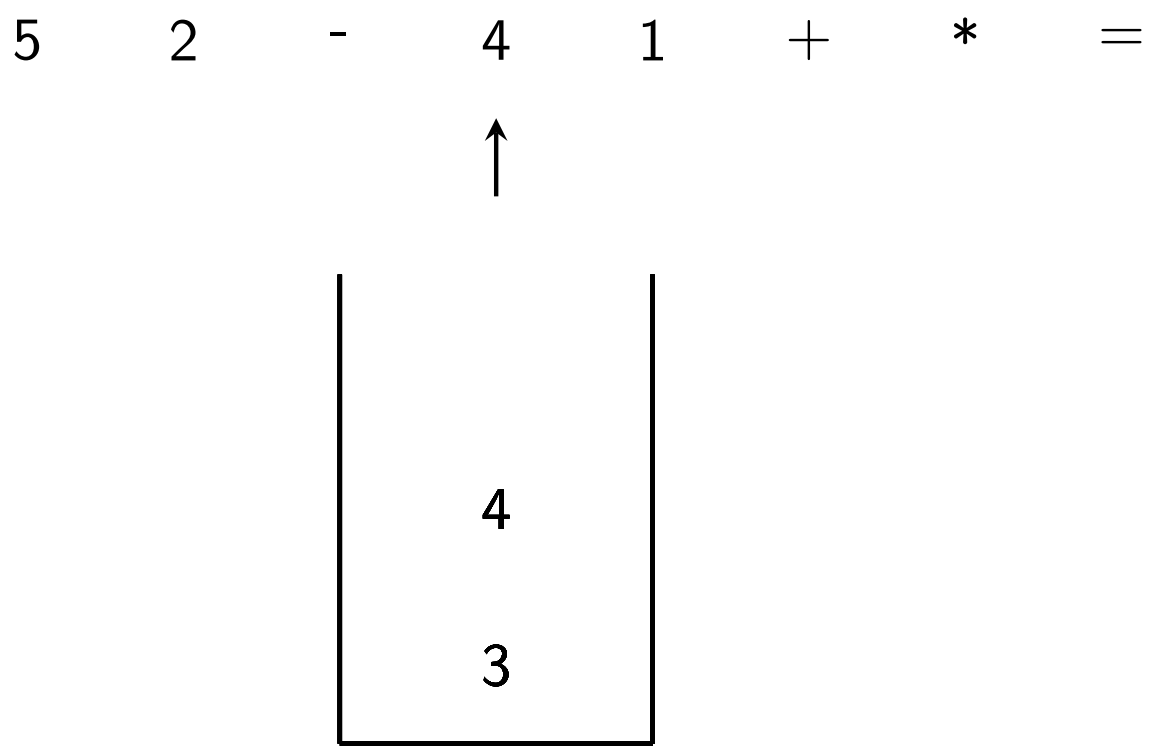
Zamieszczona poniżej animacja pokazuje związki między wyrażeniami w ONP, a stosem i obrazuje jak wyliczyć wartość takiego wyrażenia z jego użyciem.



Stos - zastosowania

Obliczanie wyrażeń w ONP

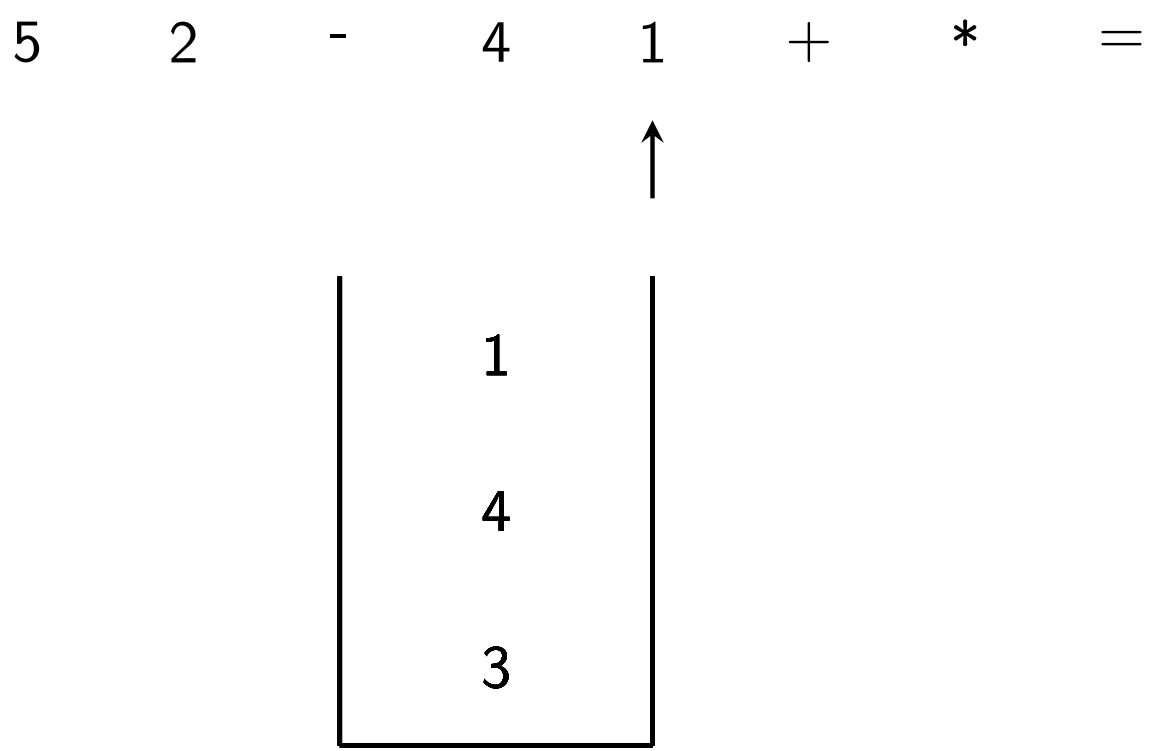
Zamieszczona poniżej animacja pokazuje związki między wyrażeniami w ONP, a stosem i obrazuje jak wyliczyć wartość takiego wyrażenia z jego użyciem.



Stos - zastosowania

Obliczanie wyrażeń w ONP

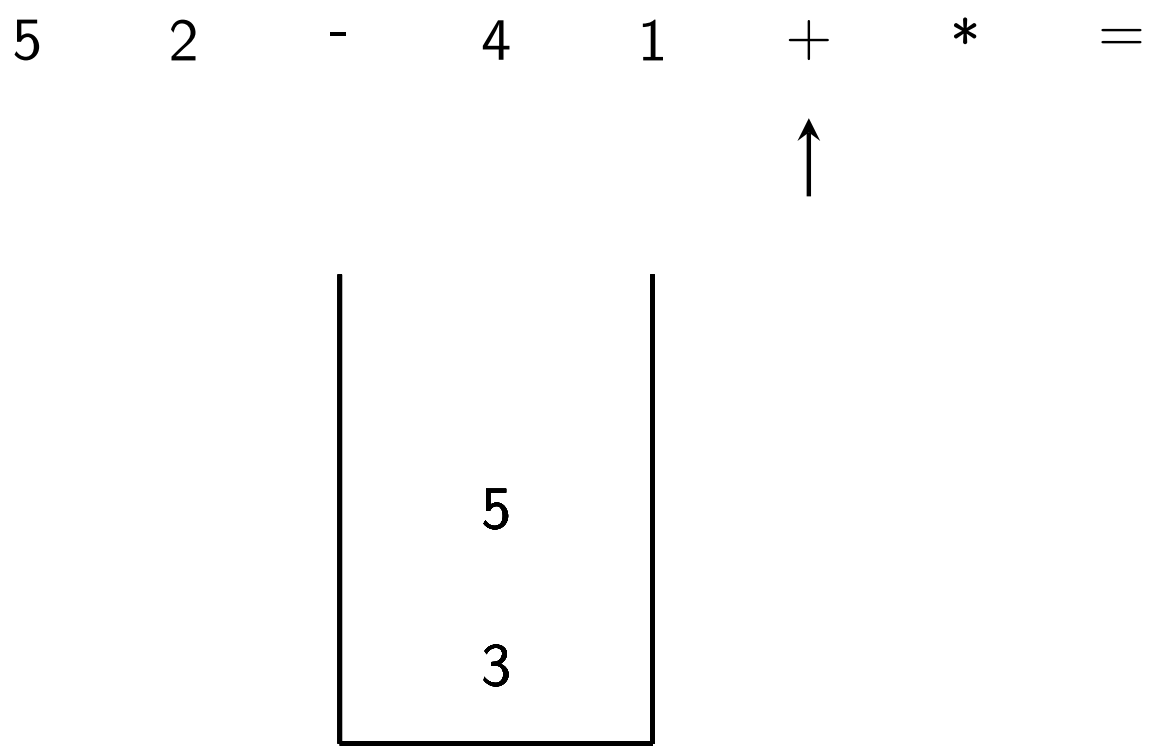
Zamieszczona poniżej animacja pokazuje związki między wyrażeniami w ONP, a stosem i obrazuje jak wyliczyć wartość takiego wyrażenia z jego użyciem.



Stos - zastosowania

Obliczanie wyrażeń w ONP

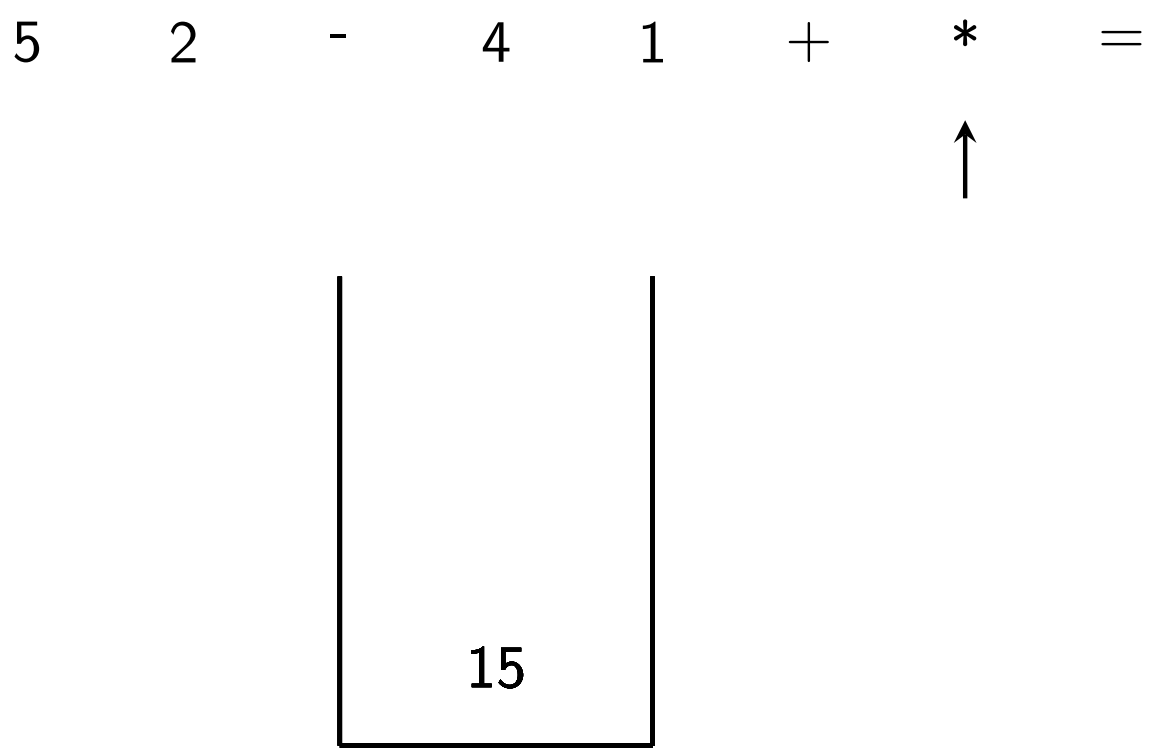
Zamieszczona poniżej animacja pokazuje związki między wyrażeniami w ONP, a stosem i obrazuje jak wyliczyć wartość takiego wyrażenia z jego użyciem.



Stos - zastosowania

Obliczanie wyrażeń w ONP

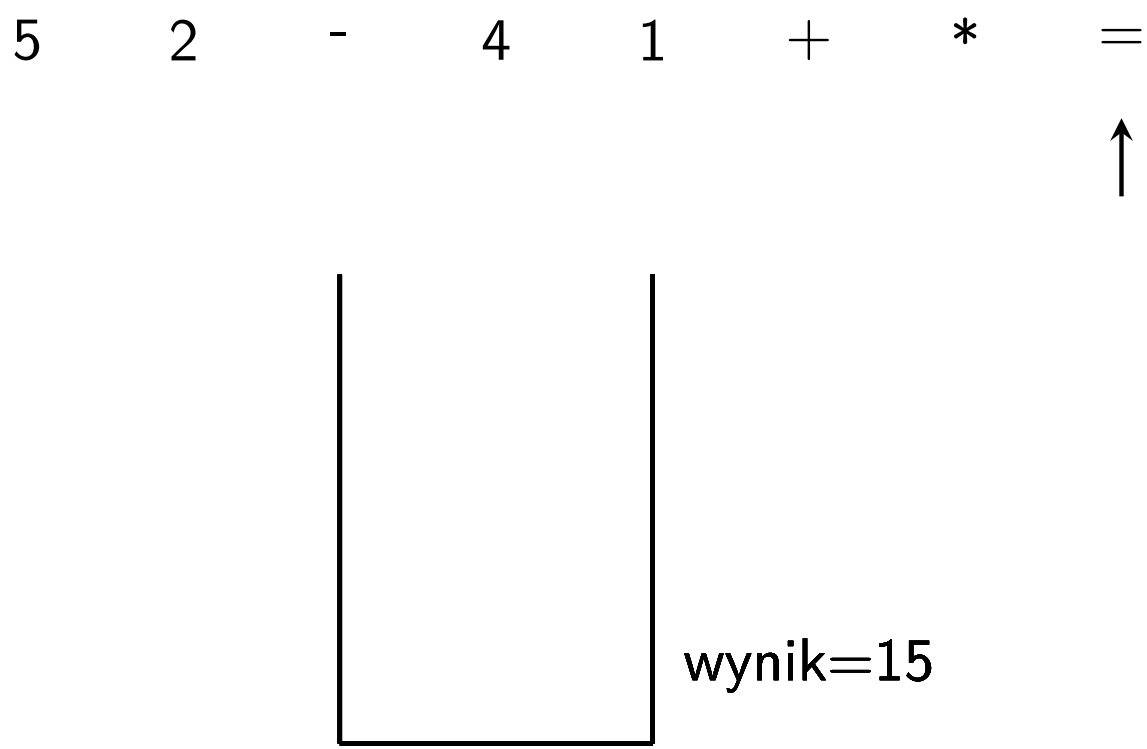
Zamieszczona poniżej animacja pokazuje związki między wyrażeniami w ONP, a stosem i obrazuje jak wyliczyć wartość takiego wyrażenia z jego użyciem.



Stos - zastosowania

Obliczanie wyrażeń w ONP

Zamieszczona poniżej animacja pokazuje związki między wyrażeniami w ONP, a stosem i obrazuje jak wyliczyć wartość takiego wyrażenia z jego użyciem.



Stos - zastosowania

Obliczanie wyrażeń w ONP

Z zamieszczonej na poprzedniej stronie animacji wynika, że wyrażenie w ONP jest czytane od lewej strony. Jeśli napotkany jego element jest liczbą, to jest ona odkładana na stos, a jeśli operatorem, to ze stosu pobierane są jego argumenty (jeśli wyrażenie jest poprawne, to ich odpowiednia liczba będzie na stosie), wykonywane jest działanie, a jego wynik umieszczany jest z powrotem na stosie. Pisząc program przyjmiemy następujące upraszczające problem założenia:

1. wyrażenie składa się wyłącznie z naturalnych liczb jednocyfrowych oraz trzech operacji: dodawania, mnożenia i odejmowania,
2. wyrażenie nie zawiera znaków białych, w tym spacji,
3. znak = kończy wyrażenie i nakazuje programowi podać jego wartość,
4. program nie kontroluje poprawności tego wyrażenia - zakłada, że jest ono prawidłowe.

Stos - zastosowania

Obliczanie wyrażeń w ONP

Początek programu jest taki sam, jak w przypadku program konwertującego zapis liczby z binarnego na dziesiętny - tutaj również używamy biblioteki *curses* oraz funkcji `push()` i `pop()`.

Stos - zastosowania

Obliczanie wyrażeń w ONP

```
#include<stdlib.h>
#include<curses.h>
#include<locale.h>

struct stack_node {
    int data;
    struct stack_node *next;
};
```

Stos - zastosowania

Obliczanie wyrażeń w ONP

```
struct stack_node *push(struct stack_node *top, int number)
{
    struct stack_node *new_node = (struct stack_node *)
                          malloc(sizeof(struct stack_node));
    if(new_node!=NULL) {
        new_node->data = number;
        new_node->next = top;
        top = new_node;
    }
    return top;
}
```

Stos - zastosowania

Obliczanie wyrażeń w ONP

```
int pop(struct stack_node **top)
{
    int result = -1;
    if(*top) {
        result = (*top)->data;
        struct stack_node *tmp = (*top)->next;
        free(*top);
        *top = tmp;
    }
    return result;
}
```

Stos - zastosowania

Obliczanie wyrażeń w ONP

```
int calculate_rpn_expression(void)
{
    int input = 0;
    struct stack_node *top = NULL;
    do {
        input = getch();
        int first_argument=0, second_argument=0, result=0;
        switch(input) {
        case '+':
            result = pop(&top) + pop(&top);
            top = push(top,result);
            break;
        case '-':
            first_argument = pop(&top);
            second_argument = pop(&top);
            top = push(top,second_argument - first_argument);
            break;
```

Stos - zastosowania

Obliczanie wyrażeń w ONP

```
            case '*':
                result = pop(&top)*pop(&top);
                top = push(top,result);
                break;
            default:
                if(input>='0'&&input<='9')
                    top=push(top,input-'0');
            }
        } while(input!='=');

        return pop(&top);
    }
```

Stos - zastosowania

Obliczanie wyrażeń w ONP

Zaprezentowana na dwóch poprzednich slajdach bezparametrowa funkcja `calculate_rpn_expression()`, pobiera w pętli `do...while` od użytkownika kolejne znaki należące do wyrażenia w ONP i je rozpoznaje. Rozpoznawanie wyrażeń w informatyce nazywa się *parsowaniem*. Jeśli odczytany z klawiatury znak jest cyfrą, to program zapisuje jej wartość liczbową na stosie. Jeśli znak jest którymś z obsługiwanych działań, to funkcja pobiera ze stosu dwie liczby[1], wykonuje rozpoznane działanie i zapisuje jego wynik z powrotem na stosie. Pętla kończy się po napotkaniu znaku „równa się". Po jej zakończeniu funkcja zwraca ostatnią wartość zapisaną na stosie i kończy swoje działanie. Po zakończeniu funkcji stos powinien być pusty.

[1] **Uwaga na ich kolejność przy odejmowaniu!**

Stos - zastosowania

Obliczanie wyrażeń w ONP

```
int main(void)
{
    if(setlocale(LC_ALL,"")==NULL) {
        fprintf(stderr,"Błąd inicjacji ustawień językowych!\n");
        return -1;
    }
    if(!initscr()) {
        fprintf(stderr,"Błąd inicjacji biblioteki curses!\n");
        return -1;
    }
    printw("Wprowadź wyrażenie w Odwrotnej Notacji Polskiej\n");
    (void)refresh();
    printw("\nWynik działania to: %d.\n",calculate_rpn_expression());
    (void)refresh();
    getch();
    if(endwin()==ERR) {
        fprintf(stderr,"Błąd funkcji endwin()!\n");
        return -1;
    }
    return 0;
}
```

Stos - zastosowania

Obliczanie wyrażeń w ONP

W funkcji `main()` programu, poza instrukcjami inicjującymi i wyłączającymi pracę biblioteki *curses* oraz zmieniającymi ustawienia językowe, wywoływana jest funkcja `calculate_rpn_expression()`. Zwracaną przez nią wartością jest wynik obliczenia wartości wrażenia arytmetycznego w ONP. Długość wyrażenia, dzięki zastosowaniu stosu w postaci dynamicznej struktury danych, jest ograniczona wielkością sterty, natomiast jego wartość musi się mieścić w zakresie typu `int`.

Stos - zastosowania

Stos łańcuchów znaków

Ostatni przykład pokazuje jak użyć stosu do przechowywania ciągów znaków. Zakładamy w nim, że liczba znaków w pojedynczym łańcuchu nie będzie przekraczała 100. Program nie korzysta z biblioteki *curses*. Ciągi, które mają być przez niego zapisane na stosie są mu przekazywane jako argumenty jego wywołania. W programie oprócz plików nagłówkowych `stdio.h` i `stdlib.h` włączono także plik do obsługi łańcuchów znaków.

Stos - zastosowania

Stos łańcuchów znaków

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

typedef struct stack_node {
    char arg[100];
    struct stack_node *next;
} node;
```

Stos - zastosowania

Stos łańcuchów znaków

W programie została zmieniona deklaracja pierwszego pola struktury, tak aby mogło ono przechowywać ciągi składające się z co najwyżej 100 znaków. Zmieniona została także jego nazwa. Zastosowano również słowo kluczowe `typedef`, aby uniknąć pisania całej specyfikacji typu wskaźnikowego w definicjach funkcji. Jest to wygodne rozwiązanie, ale niestety trochę mniej czytelne.

Stos - zastosowania

Stos łańcuchów znaków

```
node *push(node *top, char *str)
{
    node *new_node = NULL;
    new_node = (node*)malloc(sizeof(node));
    if(new_node==NULL) {
        fprintf(stderr,"Błąd przydziału pamięci\n");
        return top;
    }
    strncpy(new_node->arg,str,100);
    new_node->next=top;
    return new_node;
}
```

Stos - zastosowania

Stos łańcuchów znaków

Funkcja `push()` jest zdefiniowana podobnie jak w poprzednich przykładach, ale teraz jako drugi argument przyjmuje wskaźnik na ciąg znaków. Ten ciąg znaków jest kopiowany za pomocą `strncpy()` do nowego elementu stosu. Proszę także zwrócić uwagę na to, że funkcja wypisuje komunikat w przypadku niepowodzenia operacji przydziału pamięci i dopiero po tej czynności zwraca przekazany jej wskaźnik stosu. Także zwracanie nowego wskaźnika zostało trochę krócej zapisane - po połączeniu nowego elementu do stosu jest zwracany jego adres, który w miejscu wywołania funkcji powinien być przypisany do wskaźnika stosu.

Stos - zastosowania

Stos łańcuchów znaków

```
node *pop(node **top)
{
    node *next = NULL, *old_top = NULL;
    if(*top!=NULL) {
        next=(*top)->next;
        (*top)->next = NULL;
        old_top = *top;
        *top = next;
    }
    return old_top;
}
```

Stos - zastosowania

Stos łańcuchów znaków

Zaprezentowana na poprzednim slajdzie implementacja funkcji `pop()` nie zwalnia pamięci przeznaczonej na element szczytowy stosu, a jedynie odłącza go od reszty stosu, zeruje jego pole `next` i zwraca jego adres. Do funkcji przez parametr będący podwójnym wskaźnikiem przekazywany jest wskaźnik na stos, który jest tam modyfikowany, tzn. po odłączeniu elementu szczytowego, do tego wskaźnika zapisywany jest adres elementu, który teraz znalazł się na szczycie stosu.

Stos - zastosowania

Stos łańcuchów znaków

```
int main(int argc, char **argv)
{
    node *top = NULL;
    int i;
    for(i=0; i<argc; i++)
        top = push(top,argv[i]);
    while(top) {
        node *tmp = pop(&top);
        printf("%s\n",tmp->arg);
        free(tmp);
        tmp=NULL;
    }
    return 0;
}
```

Stos - zastosowania

Stos łańcuchów znaków

W funkcji `main()` argumenty wywołania programu są zapisywane na stosie wewnątrz pętli `for`. Ponieważ argument zapisywany w elemencie tablicy `argv` o indeksie `0` zawsze istnieje - jest to pełna ścieżka do pliku wykonywalnego programu, to program zawsze wypisze na ekran przynajmniej jeden komunikat. Elementy ze stosu są zdejmowane w pętli `while`. Proszę zwrócić uwagę, że są one pojedynczo niszczone poza funkcją `pop()`.

Podsumowanie

Stos może mieć wiele więcej zastosowań, niż te pokazane wcześniej. W kompilatorach jest używany do wyznaczania wartości wyrażeń arytmetycznych. Algorytm konwersji takich wyrażeń z notacji wrostkowej na przyrostkową również wykorzystuje stos. Tym razem w jego elementach są zapisywane jednak operatory i nawiasy. Autorem tego algorytmu jest Edsgar Dijkstra, a nosi on nazwę algorytmu stacji rozrządowej (ang. *shunting-yard algorithm*). Nie będzie on tu jednak dokładniej opisywany. W systemach operacyjnych stos może służyć do zarządzania procesami i zasobami. W programach użytkowych, takich jak edytory tekstu ta struktura stosowana jest do realizacji operacji „cofnij” (ang. *undo*).

Podsumowanie

Stos może być zrealizowany za pomocą zwykłej tablicy, ale wtedy jego pojemność jest z góry ograniczona wielkością tej struktury danych. Jako wskaźnik stosu można wykorzystać w takiej implementacji indeks. Dodanie elementu do stosu jest równoważne zwiększeniu wartości tego indeksu o jeden i zapisaniu danej do elementu tablicy przez niego określanego. Usunięcie elementu ze stosu polega na odczycie danej z elementu określanego przez indeks i zmniejszeniu wartości tego indeksu o jeden.

Pytania

?

KONIEC

Dziękuję Państwu za uwagę!