Informatyka 2

Politechnika Białostocka - Wydział Elektryczny

Elektrotechnika, semestr III, studia stacjonarne I stopnia

Rok akademicki 2008/2009

Wykład nr 2 (22.10.2008)

dr inż. Jarosław Forenc

Plan wykładu nr 2

- Dynamiczne struktury danych
  - przykład zastosowania stosu - notacja polska i odwrotna notacja polska
  - kolejka
  - lista (jednokierunkowa, dwukierunkowa, cykliczna)
  - drzewo (binarne)
- Operacje wejścia/wyjścia w C++
- Programowanie proceduralne i obiektowe
- Programowanie obiektowe w języku C++
  - definicja klasy, składniki klasy: dane i funkcje
  - prawa dostępu do składników klasy
  - definiowanie funkcji składowych wewnątrz i poza klasą

Notacja polska

- notacja polska (zapis przedrostkowy, Notacja Łukasiewicza) jest to sposób zapisu wyrażeń arytmetycznych, podający najpierw operator, a następnie argumenty
- wyrażenie arytmetyczne: **`4 / (1 + 3)`**
  ma w notacji polskiej postać: **`/ 4 + 1 3`**
- wyrażenie w notacji polskiej nie wymaga nawiasów, ponieważ przypisanie argumentów do operatorów wynika wprost z ich kolejności w zapisie
- notacja polska jest bliska naturalnemu sposobowi wyrażania działań, w którym zazwyczaj najpierw podaje się czynność, a następnie dopełnia wyrażenia wskazaniem rzeczy, do których czynność się odnosi, np.

  „podziel cztery przez sumę jednego i trzech”

- notację polską przedstawił w 1920 roku polski matematyk Jan Łukasiewicz
- zapis wyrażeń w notacji polskiej stał się podstawą języków: Logo, Tcl i LISP
- notacja polska była podstawą opracowania tzw. odwrotnej notacji polskiej

Odwrotna notacja polska

- odwrotna notacja polska - ONP (ang. Reverse Polish Notation, RPN) jest sposobem zapisu wyrażeń arytmetycznych, w którym znak wykonywanej operacji umieszczany jest po argumentach, a nie pomiędzy nimi jak w konwencjonalnym zapisie algebraicznym

- wyrażenie arytmetyczne: **(1 + 3) / 2**
  ma w odwrotnej notacji polskiej postać: **1 3 + 2 /**

- ONP została opracowana przez australijskiego naukowca Charlesa Hamblina jako „odwrócenie" beznawiasowej notacji polskiej Jana Łukasiewicza na potrzeby zastosowań informatycznych
- zapis wyrażenia w ONP pozwala na całkowita rezygnację z użycia nawiasów w wyrażeniach, gdyż jednoznacznie określa kolejność wykonywania działań
- ONP używana jest w niektórych językach programowania (FORTH, Postscript) oraz w kalkulatorach naukowych HP:
  - programy komputerowe kompilujące program analizują wyrażenie arytmetyczne i przekształcają je na ciąg instrukcji odpowiadający odwrotnej notacji polskiej
  - otrzymane wyrażenie obliczane jest podczas wykonywania programu

Odwrotna notacja polska

- obliczenie wartości wyrażenia arytmetycznego przy zastosowaniu odwrotnej notacji polskiej wymaga wykonania dwóch operacji:
  - zamiany notacji konwencjonalnej (nawiasowej) na odwrotną notację polską
  - obliczenia wartości wyrażenia arytmetycznego zapisanego w odwrotnej notacji polskiej
- oba powyższe algorytmy są bardzo proste i wykorzystują stos

Zamiana wyrażenia z notacji konwencjonalnej na ONP:

- zamiana wykonywana jest przy zastosowaniu algorytmu Dijkstry nazywanego stacją rozrządową
- czytając wyrażenie arytmetyczne od strony lewej do strony prawej operatory odkładamy na stos a liczby na wyjście
- wyjście należy traktować jako kolejkę, która po zakończeniu algorytmu będzie zawierała wyrażenie w odwrotnej notacji polskiej

Odwrotna notacja polska

Zamiana wyrażenia z notacji konwencjonalnej na ONP:

- wykonując powyższe operacje trzeba stosować następujące reguły:
  - operator możemy odłożyć na stos tylko wtedy, jeśli ostatnim elementem stosu jest operator o niższym priorytecie
  - jeżeli ma on wyższy lub równy priorytet to zdejmujemy ze stosu dotąd elementy i wysyłamy na wyjście, aż ostatni operator będzie miał niższy priorytet lub stos będzie pusty
  - jeśli kolejnym elementem jest nawias otwierający „(”, to odkładamy go na stos, bez względu na to co znajduje się w danym momencie na stosie i bez względu na to czy stos jest pusty
  - powyższy nawias traktujemy jak dno stosu i odczytujemy kolejne elementy wyrażenia według standardowego algorytmu
  - jeśli dojdziemy do nawiasu zamykającego „)”, to nigdzie go nie odkładamy, tylko zdejmujemy kolejne operatory ze stosu i wysyłamy na wyjście, aż dojdziemy do nawiasu otwierającego, który również zdejmujemy ze stosu i wysyłamy na wyjście
  - jeśli dojdziemy do końca wyrażenia arytmetycznego, to zdejmujemy ze stosu pozostałe operatory i wysyłamy je na wyjście

Odwrotna notacja polska

Zamiana wyrażenia z notacji konwencjonalnej na ONP - przykład:

- równanie w notacji konwencjonalnej: **(2+1)*3-4*(7+4)**

Krok	Wejście	Stos	Wyjście
1	(	(NULL	
2	2	(NULL	2
3	+	+ (NULL	
4	1	+ (NULL	1
5	)	NULL	+
6	*	* NULL	
7	3	* NULL	3
8	-	- NULL	*
9	4	- NULL	4
10	*	* - NULL	
11	(	(* - NULL	
12	7	(* - NULL	7
13	+	+ (* - NULL	
14	4	+ (* - NULL	4
15	)	* - NULL	+
16	Koniec	- NULL	*
17		NULL	-

- równanie w ONP: **2 1 + 3 * 4 7 4 + * -**

Odwrotna notacja polska

Obliczenie wartości wyrażenia arytmetycznego w ONP:

- w algorytmie obliczania wartości wyrażenia arytmetycznego zapisanego w odwrotnej notacji polskiej wykonujemy następujące operacje:
  - pobieramy kolejny element wyrażenia
  - jeśli elementem jest liczba to odkładamy ją na stos
  - jeśli elementem jest operator, to pobieramy ze stosu tyle liczb, aby można było „zastosować" operator na tych liczbach, np. dla dodawania, odejmowania, mnożenia i dzielenia są to dwie kolejne liczby, zaś dla negacji - jedna liczba
  - wykonujemy operację na liczbach i jej wynik odkładamy na stos
  - jeśli dotrzemy do końca wyrażenia, to pobieramy wynik ze stosu, który jest wartością wyrażenia arytmetycznego
  - jeśli nie ma jeszcze końca, to wracamy na początek algorytmu

Uwaga:

- jeśli np. stos ma postać: **`2 4 NULL`** i mamy wykonać operację dzielenia **`/`**, to operacja ta ma postać: **`4 / 2`**, czyli do wykonania operacji argumenty brane są w odwrotnej kolejności

Odwrotna notacja polska

Obliczenie wartości wyrażenia arytmetycznego w ONP - przykład:

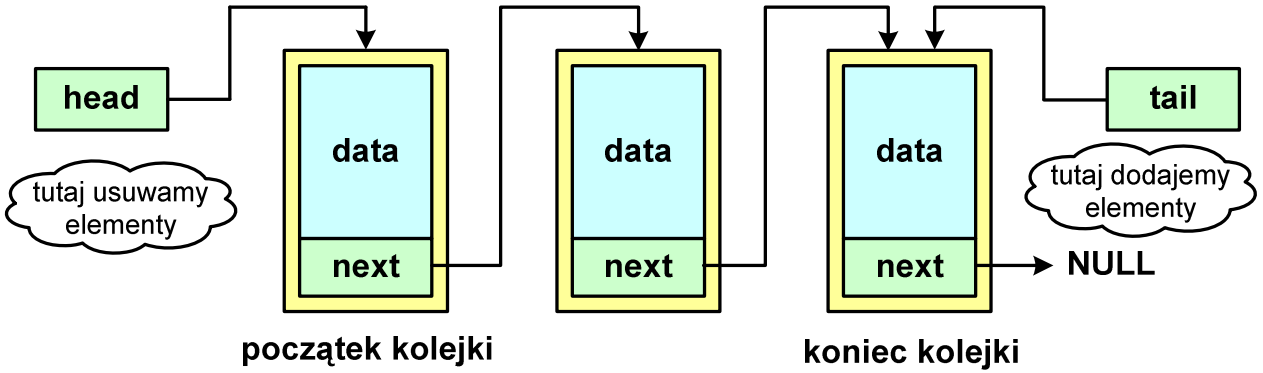
- równanie w ONP: **2 1 + 3 * 4 7 4 + * -**

Krok	Wejście	Stos	Działanie
1	2	2 NULL	
2	1	1 2 NULL	
3	+	3 NULL	2 + 1
4	3	3 3 NULL	
5	*	9 NULL	3 * 3
6	4	4 9 NULL	
7	7	7 4 9 NULL	
8	4	4 7 4 9 NULL	
9	+	11 4 9 NULL	7 + 4
10	*	44 9 NULL	4 * 11
11	-	-35 NULL	9 – 44
12	Koniec	NULL	wynik: -35

- wynik: **-35**

Kolejka

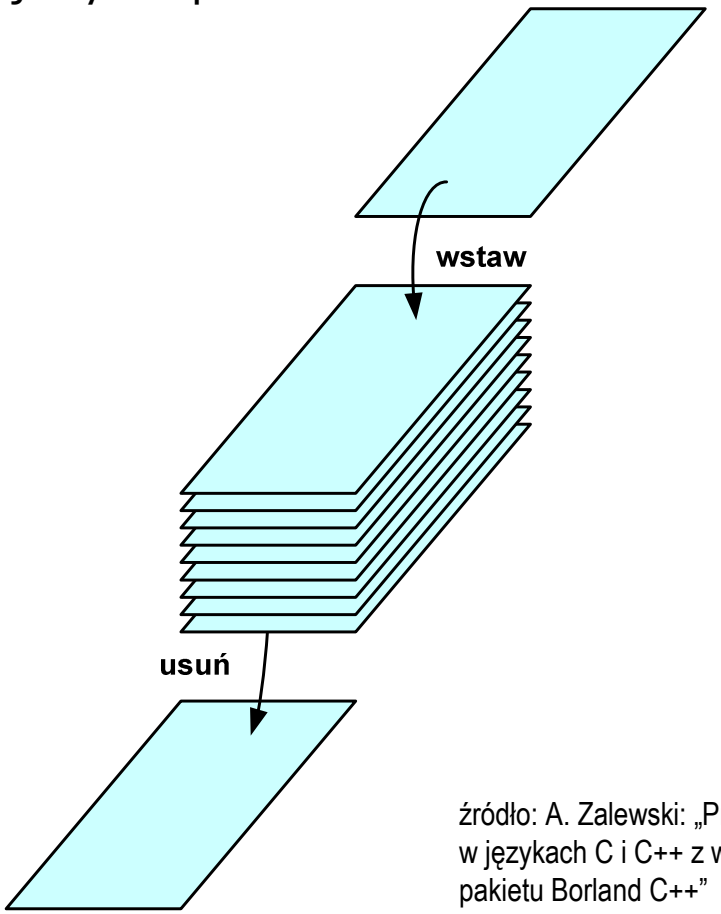
- kolejka (ang. queue) jest strukturą danych składającą się z liniowo uporządkowanych elementów, do której można dołączać elementy tylko w jednym końcu (na końcu kolejki), a usuwać tylko w drugim końcu (na początku kolejki)
- kolejka często określana jest jako stos FIFO (ang. First In First Out - pierwszy wchodzi, pierwszy wychodzi)
- powiązanie między elementami kolejki jest takie samo, jak w przypadku stosu



- head jest wskaźnikiem na pierwszy element kolejki (początek kolejki), zaś tail - na ostatni element kolejki (koniec kolejki)

Kolejka

- korzystając z poprzedniej analogii, stosu kartek, możemy powiedzieć, że kładziemy kartki na wierzchołku stosu, zaś wyjmujemy ze spodu
- kolejkę można także wyobrażać sobie jako typową kolejkę sklepową
- podstawowe operacje dotyczące kolejki to:
  - dołączenie elementu do kolejki - Insert() lub enqueue()
  - usunięcie elementu z kolejki - Remove() lub dequeue()



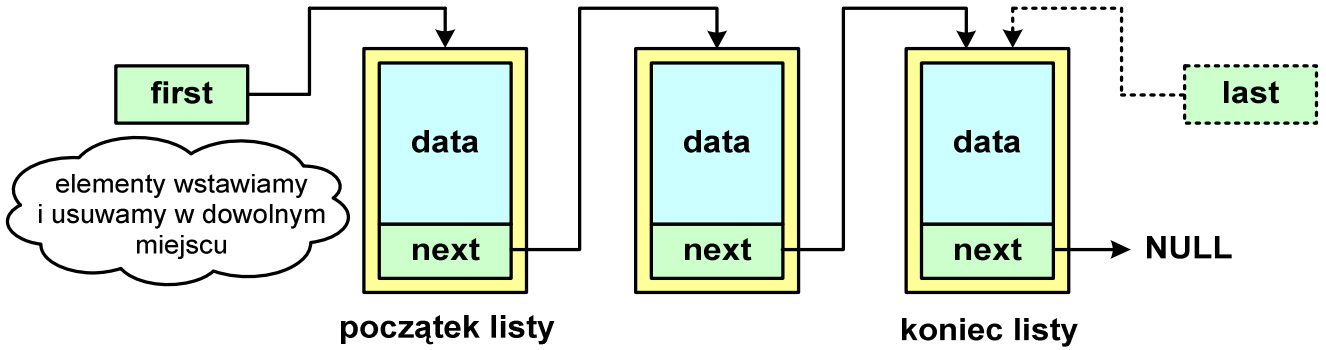
źródło: A. Zalewski: „Programowanie w językach C i C++ z wykorzystaniem pakietu Borland C++”

Lista

- listą (liniową) nazywamy liniowo uporządkowany zbiór elementów, z którego w dowolnym miejscu można usunąć element, jak również dołączyć nowy element
- sposób budowy listy jest ściśle uzależniony od zamierzeń programisty
- elementy można wstawiać do listy na początku, na końcu lub w dowolnym innym miejscu (np. w celu przechowywania elementów posortowanych)
- w zależności od powiązań pomiędzy elementami wyróżniamy listy:
  - jednokierunkowe
  - dwukierunkowe
  - cykliczne, jednokierunkowe
  - cykliczne, dwukierunkowe

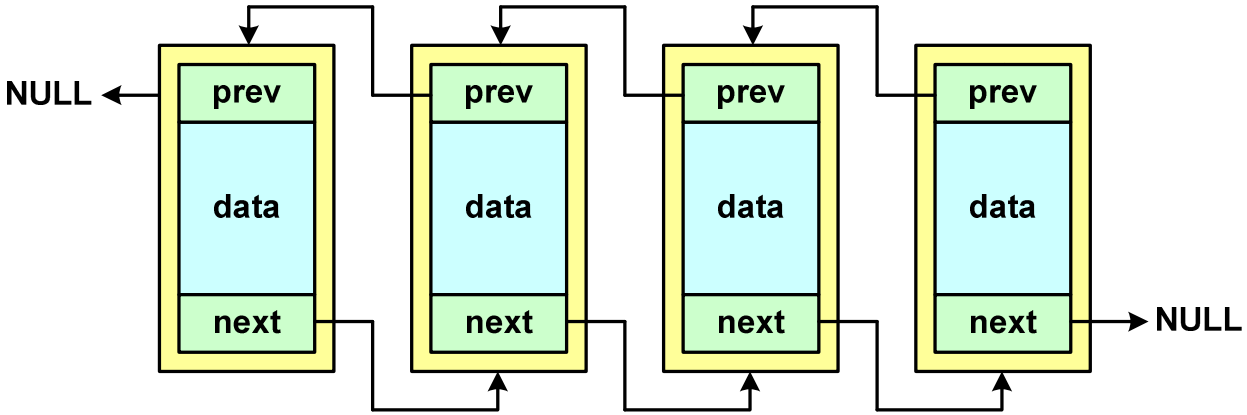
Lista jednokierunkowa

- organizacja listy jednokierunkowej podobna jest do organizacji stosu i kolejki, tzn. dla każdego składnika (poza ostatnim) jest określony następny składnik (lub poprzedni - zależnie od implementacji)
- zapamiętywany jest wskaźnik tylko na pierwszy element listy lub wskaźniki na pierwszy i ostatni element listy



Lista dwukierunkowa

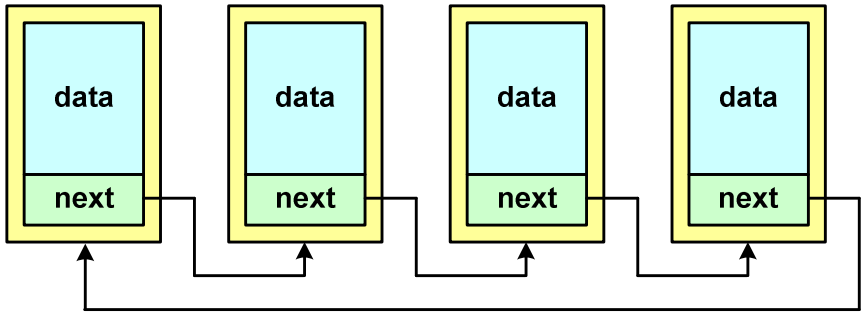
- w liście dwukierunkowej każdy węzeł posiada adres następnika, jak i poprzednika
- w strukturze tego typu wygodne jest przechodzenie pomiędzy elementami w obu kierunkach (od początku do końca i odwrotnie)



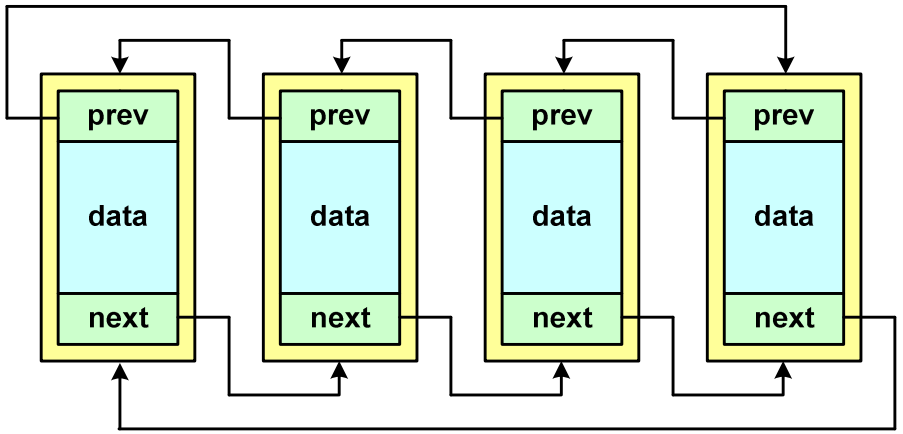
Lista cykliczna

- listę cykliczną można utworzyć z listy jednokierunkowej lub dwukierunkowej, jeśli ostatni element tej struktury połączymy z pierwszym

Jednokierunkowa:

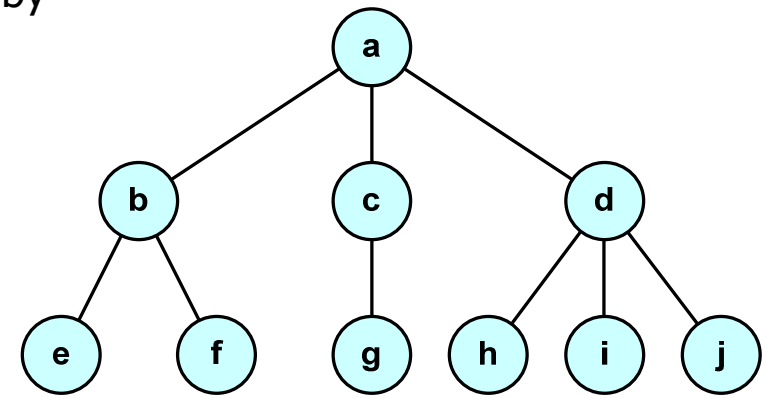


Dwukierunkowa:



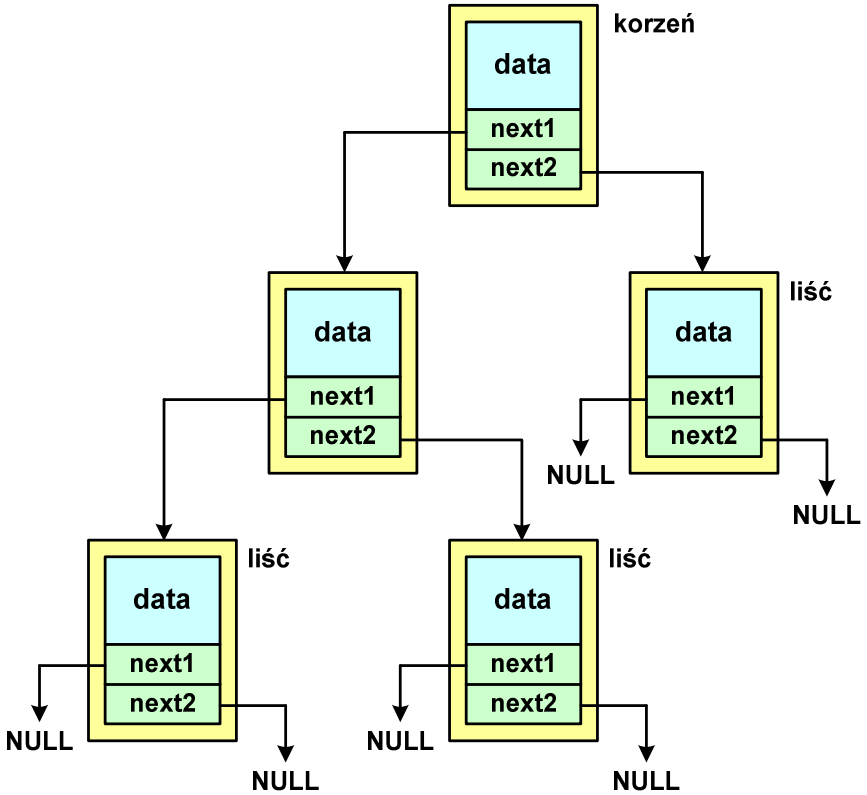
Drzewo

- drzewo jest najbardziej ogólną dynamiczną strukturą danych i może być reprezentowane graficznie na różne sposoby
- korzeń drzewa jest umieszczony u góry
- skojarzone z korzeniem poddrzewa połączone są z korzeniem liniami zwanymi gałęziami drzewa
- potomkiem węzła w nazywamy każdy, różny od w, węzeł należący do drzewa, w którym w jest korzeniem
- węzeł w nazywamy przodkiem węzłów drzewa, w którym w jest korzeniem
- bezpośrednich potomków nazywamy synami
- bezpośrednich przodków nazywamy ojcami
- synów tego samego ojca nazywamy braćmi
- węzeł, który nie ma potomków, to liść drzewa



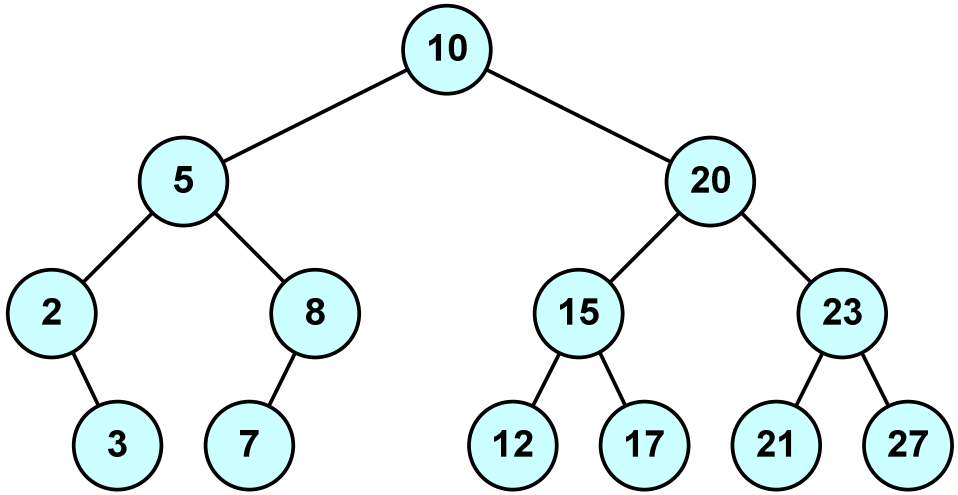
Drzewo binarne

- drzewo binarne jest szczególnym przypadkiem ogólnej struktury zwanej drzewem
- każdy wierzchołek drzewa ma co najwyżej dwóch potomków, tzn. każdy ojciec ma co najwyżej dwóch synów



Binarne drzewo wyszukiwawcze

- jest to drzewo binarne, w którym dla każdego węzła w_i wszystkie klucze (przechowywane wartości) w lewym poddrzewie węzła w_i są mniejsze od klucza w węźle w_i, a wszystkie klucze w prawym poddrzewie węzła w_i są większe od klucza w węźle w_i
- największą zaletą takiej struktury jest szybkość wyszukiwania informacji



Operacje wejścia/wyjścia w języku C++

- operacje wejścia-wyjścia nie są zdefiniowane w języku C++ - umożliwiają je biblioteki standardowo dołączane przez producenta kompilatora:

  1) biblioteka stdio (język C)
  2) biblioteka stream (stara wersja iostream)
  3) biblioteka iostream

Strumienie:

- wprowadzanie i wyprowadzanie informacji można potraktować jako strumień bajtów płynących od źródła do ujścia.
- strumienie w C++ realizowane są na zasadzie klas
- wykorzystanie strumieni wymaga dołączenia pliku nagłówkowego iostream

  #include <iostream> zamiast #include <stdio.h>

Operacje wejścia/wyjścia w języku C++

Strumienie:

- w języku C++ są cztery predefiniowane strumienie:

  cout - związany ze standardowym urządzeniem wyjścia (ekran), skrót od ang. C-onsole OUT-put

  cin - związany ze standardowym urządzeniem wejścia (klawiatura), skrót od ang. C-onsole IN-put

  cerr - związany ze standardowym urządzeniem, na które chce się wypisywać komunikaty o błędach (ekran) - strumień niebuforowany

  clog - związany ze standardowym urządzeniem, na które chce się wypisywać komunikaty o błędach (ekran) - strumień buforowany

Operacje wejścia/wyjścia w języku C++

Operatory << i >>:

- `<<` - operator odpowiadający za wysyłanie informacji do strumienia, nazywany jest często operatorem insert - wstawienia (albo put to)
- `>>` - operator odpowiadający za wczytywanie informacji, nazywany jest operatorem ekstrakcji (extract operator) lub operatorem get from

Przykłady:

`cout << x;`	- wyświetlenie na ekranie wartości zmiennej x
`cout << x << y;`	- wyświetlenie wartości dwóch zmiennych: x i y (bez znaku spacji między nimi)
`cout << x << " " << y;`	- wyświetlenie wartości dwóch zmiennych: x i y (ze spacją między nimi)
`cin >> x;`	- wczytanie informacji do zmiennej x

- operatory << i >> mają bardzo niski priorytet

Program w języku C++

Przykład:

```
#include <iostream>

int main()
{
   std::cout << "Witaj swiecie!" << std::endl;
   system("pause");
}
```

- std:: przed nazwami identyfikatorów cout i endl oznacza, że pochodzą one z biblioteki standardowej (dokładniej - pochodzą z tzw. przestrzeni nazw std)
- endl - przejście do nowego wiersza, odpowiada "\n" w języku C
- w celu uniknięcia ciągłego pisania std:: przed nazwami identyfikatorów umieszcza się w programie dyrektywę using: using namespace std;

Program w języku C++

Przykład (bez dyrektywy using):

```
#include <iostream>

int main()
{
   std::cout << "Witaj swiecie!" << std::endl;
   system("pause");
}
```

Przykład (z dyrektywą using):

```
#include <iostream>
using namespace std;

int main()
{
   cout << "Witaj swiecie!" << endl;
   system("pause");
}
```

Techniki programowania

Programowanie liniowe:

- instrukcje programu umieszczone są jedna za drugą, na górze jest początek programu, na dole - jego koniec
- wszystkie zmienne są globalne, brak jest zmiennych lokalnych
- bardzo często występują instrukcje skoku goto - z tego powodu analiza programu jest bardzo trudna
- przykład: język BASIC bez instrukcji GOSUB

Programowanie proceduralne:

- w programie wyodrębnia się części realizujące pewne określone czynności (funkcje)
- napisanie programu polega na określeniu jakie funkcje będą potrzebne, następnie zdefiniowaniu ich i wymyśleniu najlepszego algorytmu ich wywołania
- wykonanie programu staje się zatem określoną sekwencją wywołań różnych funkcji
- w ramach funkcji można definiować zmienne lokalne, ale dane występujące w programie nie są ze sobą powiązane
- przykład: język FORTAN

Techniki programowania

Programowanie z ukrywaniem danych:

- zastosowanie w programie struktur (rekordów) pozwalających grupować pod jedną nazwą elementy różnych typów
- dane, które są ze sobą powiązane w życiu są też powiązane w programie

Programowanie obiektowe:

- do zgrupowanych danych dodaje się metody postępowania z nimi (funkcje)
- danym zgrupowanym w obiekt nie mówi się jak mają coś zrobić, ale tylko co mają robić (jak mają coś zrobić wiedzą na podstawie swoich funkcji składowych)
- wada: poszczególne typy danych są sobie „obce” - jeśli funkcja jako argument może przyjąć jeden typ danej, to nie może innego; nowy typ oznacza konieczność pisania funkcji od początku

Programowanie Obiektowo Orientowane (OO):

- technika obiektowa wzbogacona o dziedziczenie i polimorfizm (funkcje wirtualne)
- kod programu jest w stanie sam „zorientować się” z jakim obiektem pracuje

Zaletą C++ jest możliwość stosowania dowolnej techniki programowania

Techniki programowania

- program komputerowy opisuje (modeluje) w pewien sposób rzeczywistość
- w odniesieniu do sposobu definiowania przedmiotów występujących w rzeczywistości stosowane są dwa zasadnicze podejścia:

Proceduralne:

- tworzymy strukturę zawierającą parametry definiujące przedmiot i jego stan, a odseparowane funkcje określają jego właściwości

```
struct punkt
{
    int x,y;
};

void rysuj(struct punkt p)
{
    // ciało funkcji
}
```

Obiektowe:

- dane definiujące przedmiot i metody (funkcje składowe) określające jego właściwości umieszczamy we wspólnym miejscu - klasie

```
class PUNKT
{
    int x,y;

 public:
    void rysuj(void);
};
```

Struktury w języku C

- struktura jest zestawem elementów różnych typów, zgrupowanych pod jedną nazwą, służą zatem do reprezentacji złożonych obiektów różnych danych
- ogólna postać deklaracji struktury jest następująca:

```
struct nazwa
{
     opis_pola_1;
     opis_pola_2;
     ...
     opis pola_n;
};
```

- deklaracja struktury rozpoczyna się od słowa kluczowego struct, po którym może występować opcjonalna nazwa struktury (etykieta struktury)
- pomiędzy nawiasami klamrowymi umieszczone są pola struktury (komponenty, składowe), mające taką samą postać jak deklaracje zmiennych w programie
- w deklaracji struktury muszą występować wszystkie nawiasy klamrowe i średniki

Struktury w języku C

Przykłady:

```
struct punkt
{
    int x;
    int y;
};
```

```
struct osoba
{
    char imie[20];
    char nazwisko[30];
    int wiek;
    int wzrost,waga;
};
```

```
struct
{
    int a,b,c;
    float d,e;
};
```

- pola jednego typu można łączyć przecinkami
- nazwy pól struktury mogą być takie same jak nazwy innych zmiennych w programie, a nawet takie same jak nazwa struktury
- deklarując strukturę wprowadzamy nowy typ danych (np. struct punkt), którym można posługiwać się tak samo jak każdym innym typem standardowym

Struktury w języku C

- po klamrze kończącej listę pól struktury może występować lista zmiennych, np.

```
struct osoba
{
    char imie[20];
    char nazwisko[30];
    int wiek;
} Kowalski, Nowak;
```

- Kowalski i Nowak są zmiennymi typu struct osoba

- w przypadku deklaracji, po której nie występuje lista zmiennych, nie następuje przydział pamięci. Jeśli podana została nazwa struktury, to zmienne można zadeklarować później, np.

```
struct osoba
{
    char imie[20];
    char nazwisko[30];
    int wiek;
};
struct osoba Kowalski, Nowak;
```

Struktury w języku C - odwołania do pól struktury

- dostęp do pól struktury możliwy jest dzięki konstrukcji typu:

```
nazwa_struktury.nazwa_pola
```

- zapisanie wartości 25 do pola wiek zmiennej strukturalnej Kowalski ma postać:

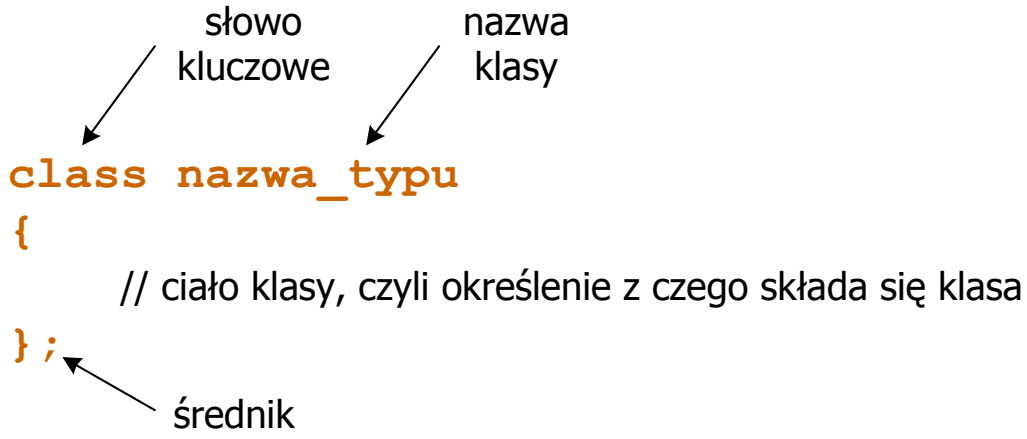
```
Kowalski.wiek = 25;
```

- operator kropki nazywany jest operatorem bezpośredniego wyboru pola
- w przypadku, gdy zmienna strukturalna jest wskaźnikiem, to do odwołania do pola struktury używamy operatora pośredniego wyboru pola (**->**), np.

```
struct osoba Nowak, *Nowak1;
Nowak1 = &Nowak
Nowak1 -> wiek = 25;        lub        (*Nowak1).wiek = 25;
```

- w ostatnim zapisie nawiasy są konieczne, gdyż operator **.** ma wyższy priorytet niż operator ***

Definicja klasy



- zmienne powyższego typu nazywa się obiektami
- jeśli chcemy utworzyć obiekt powyższej klasy, to podobnie jak przy deklaracji innych zmiennych podajemy nazwę typu i nazwę obiektu:

  nazwa_typu x; - obiekt klasy „nazwa_typu”
  nazwa_typu *y; - wskaźnik na obiekty typu „nazwa_typu”

Przykład:

```
PUNKT x, *y;
```

Składniki klasy - dane

- dane (dane składowe, pola, atrybuty) - oznaczają to samo co pola w strukturach

Przykład:

```
class osoba
{
 public:
    char imie[20];
    char nazwisko[30];
    int  wiek;
};
```

- do danych w klasie odwołujemy się w taki sam sposób jak do pól struktury:

obiekt.dana
wskaźnik->dana

```
osoba x,*y;
x.wiek = 15;
y = &x;
y->wiek = 20;
```

Składniki klasy - funkcje

- funkcje (funkcje składowe, metody) - są to funkcje operujące na danych składowych klasy

Przykład:

```
class osoba
{
   char imie[20];
   char nazwisko[30];
   int  wiek;
 public:
   void zapisz(char *i, char *n, int w);
};
```

- do funkcji w klasie odwołujemy się w taki sam sposób jak do jej danych:

  obiekt.funkcja(argumenty)
  wskaźnik->funkcja(argumenty)

```
osoba x,*y;
x.zapisz("Jan","Kowalski",30);
y = &x;
y->zapisz("Adam","Kowalski",25);
```

- deklaracje danych i funkcji mogą być umieszczane w klasie w dowolnej kolejności
- niezależnie od miejsca zdefiniowania składnika wewnątrz klasy - składnik znany jest w całej definicji klasy

Prawa dostępu do składników klasy

- dla składników klasy (danych i funkcji) określa się trzy prawa dostępu:

private (prywatne)

- oznacza, że funkcje i dane klasy dostępne są tylko z wnętrza klasy
- dla danych oznacza to, że tylko funkcje będące składnikami klasy mogą te dane odczytywać lub do nich coś zapisywać
- dla funkcji oznacza to, że mogą one zostać wywołane tylko przez inne funkcje składowe tej klasy (oraz tzw. funkcje zaprzyjaźnione)

public (publiczne)

- komponenty publiczne są ogólnie dostępne, można się do nich odwoływać z wnętrza klasy lub spoza klasy tak samo jak do pól struktur lub funkcji

protected (zabezpieczone)

- dostęp jest taki sam jak dla private, ale dodatkowo są one dostępne dla klas wywodzących się od tej klasy (dziedziczenie)

Prawa dostępu do składników klasy

- etykiety private, public, protected można umieszczać w dowolnej kolejności, mogą one powtarzać się
- domyślnie wszystkie składowe są prywatne

Przykład:

```
class osoba
{
   char imie[20];
   char nazwisko[30];
   int  wiek;
 public:
   int ocena;
   void zapisz(char *i, char *n);
 private:
   float wzrost;
   float waga;
 public:
   void wyswietl();
};
```

- definicja klasy nie definiuje obiektu - nie przydziela więc pamięci
- w definicji klasy nie można inicjować danych
- definiując kilka obiektów danej klasy w pamięci przydzielane jest miejsce dla wszystkich danych, natomiast funkcje są w pamięci tylko jeden raz
- funkcje składowe klasy mają dostęp do wszystkich jej danych i funkcji

Definiowanie funkcji składowych wewnątrz klasy

- funkcja zdefiniowana wewnątrz klasy jest funkcją inline, tzn. że podczas kompilacji wszędzie tam, gdzie występuje wywołanie funkcji zostanie wstawiony jej kod

Przykład:

```
class osoba
{
    char imie[20];
    char nazwisko[30];
    int  wiek;
 public:
    void zapisz(char *i, char *n, int w)
    {
        strcpy(imie,i);
        strcpy(nazwisko,n);
        wiek = w;
    }
    void drukuj(void)
    {
        cout << imie << " " << nazwisko;
        cout << " " << wiek << endl;
    }
};
```

- zaleca się, aby ciało funkcji zdefiniowanej wewnątrz klasy miało nie więcej niż dwie linijki kodu

Definiowanie funkcji składowych poza klasą

Przykład:

```
class osoba
{
    char imie[20];
    char nazwisko[30];
    int  wiek;
 public:
    void zapisz(char *i, char *n, int w);
    void drukuj(void);
};

void osoba::zapisz(char *i, char *n, int w)
{
    strcpy(imie,i);
    strcpy(nazwisko,n);
    wiek = w;
}

void osoba::drukuj(void)
{
    cout << imie << " " << nazwisko;
    cout << " " << wiek << endl;
}
```

deklaracje funkcji

:: - operator zakresu, pokazuje do jakiej klasy należy funkcja

Przykład nr 1 - program w języku C++ (1/2)

```
/*
  Name: klasa1.cpp
  Copyright: Politechnika Białostocka, Wydział Elektryczny
  Author: Jarosław Forenc (jarekf@pb.edu.pl)
  Date: 12-06-2007
  Description: Przykład prostej klasy
*/

#include <iostream>
#include <string.h>

using namespace std;

class osoba
{
   private:
     char imie[20];
     char nazwisko[30];
     int  wiek;
   public:
     void zapisz(char *i, char *n, int w);
     void drukuj();
};
```

Przykład nr 1 - program w języku C++ (2/2)

```
void osoba::zapisz(char *i, char *n, int w)
{
    strcpy(imie,i);
    strcpy(nazwisko,n);
    wiek = w;
}

void osoba::drukuj()
{
    cout << imie << " " << nazwisko;
    cout << " " << wiek << endl;
}

int main()
{
    osoba os1;

    os1.zapisz("Jan","Kowalski",30);
    os1.drukuj();

    system("pause");
    return 0;
}
```

```
Jan Kowalski 30
```

Koniec wykładu nr 2

Dziękuję za uwagę!