Listy i funkcje zaprzyjaźnione w C++

Na ostatnich zajęciach zajmowaliśmy się obsługą gniazdek w systemie Windows. Zajęcia te były w całości realizowane w aplikacji konsolowej. Takie podejście znaczenie upraszcza zrozumienie mechanizmów. Podobnie działać będziemy dzisiaj jednak w zupełnie odmiennym temacie. Rozmawiać będziemy o listach. Listy były już przedmiotem wykładu jak i ćwiczeń jednak dzisiaj spojrzymy na nie od innej strony.

Listy to specjalny mechanizm łączenia, który pozwala na bardzo szybkie poruszanie się po elementach. Elementem listy może być zmienna typu wbudowanego (np. int) lub też typu własnego np. struktury. To właśnie ta druga opcja pozwala na tworzenie zróżnicowanych pod względem zawartości list np. bazy danych.

Ogólnie listy można podzielić na jednokierunkowe i dwukierunkowe. Stworzenie listy polega na dodaniu co najmniej dwóch elementów z których jeden to wskaźnik pokazujący na kolejny element – obiekt tej samej struktury. Przykład:

```
struct PERSON
{
        char FirstName[20];
        char LastName[10];
        char Street[20];
        char City[20];
        PERSON* next;
};
```

Można zauważyć, że struktura *PERSON*, to złożenie danych osobowych oraz wskaźnika do kolejnego obiektu tej struktury. Aby stworzyć listę wystarczy dodać:

```
PERSON*  poczatek=new PERSON;
PERSON*  element1=new PERSON;
element1->next=NULL;
poczatek->next=element1;
```

Teraz „poczatek” jest pierwszym elementem listy a kolejnym jest „element1”. Tworzenie list jest jest proste, jednak dla jeszcze większej wygody warto wykorzystać bibliotekę „list”:

```
#include <list> //biblioteka listy
```

Przejdźmy zatem do zadania. Pierwszym ćwiczeniem jest stworzenie programu, który wyświetli nam listę obiektów typu *int*.

Zanim napiszmy nas program warto zapoznać się z podstawowymi metodami naszej biblioteki:

1. Utworzenie listy

```
list<int> first; // pusta lista elementów typu int
```

```
list<int> second (4,100); // lista czterech zmiennych typu int o wartości 100
list<int> third (second.begin(),second.end()); // tworzy listę i kopiuje zawartość listy „second” od elementu pierwszego do ostatniego
list<int> fourth (third); // tworzy listę i jednocześnie uzupełnia jej zawartość elementami zmiennej „third”
```

Można też stworzyć listę na bazie tablicy:

```
int myints[] = {16,2,77,29};
list<int> fifth (myints, myints + sizeof(myints) / sizeof(int) );
```

Aby wyświetlić listę można użyć polecenia:

```
cout << "Zawartość listy „fifth” : ";
for (list<int>::iterator it = fifth.begin(); it != fifth.end(); it++)
cout << *it << " ";
```

2. Operator podstawienia:

```
list<int> first (3); // lista trójelementowa, wartości 0
list<int> second (5); // lista pięcioelementowa, wartości 0
second=first; //teraz lista „second” jest listą trójelementową
```

3. Metoda begin();

Ustawia iterator na początek listy, natomiast metoda end() ustawia iterator na koniec listy.

Mając już podstawy możemy napisać nasz program. Dodajemy preambułę do pliku, gdzie występuje nasz *main*:

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

a następnie w *main* wpisujemy:

```
int myints[] = {75,23,65,42,13};
list<int> mylist (myints,myints+5);
list<int>::iterator it;
cout << "mylist contains:";
for ( it=mylist.begin() ; it != mylist.end(); it++ )
cout << " " << *it;
cout << endl;
```

Ostatecznie kompilujemy i uruchamiamy nasz program. W konsoli powinna zostać wyświetlona nasza lista:

```
mylist contains: 75 23 65 42 13
```

Kolejnymi interesującymi metodami są (rbegin(), rend()), które umożliwiają dostęp do elementów listy „od końca” do początku. Przykładem wykorzystania jest program:

```
#include <iostream>
#include <list>
using namespace std;
int main ()
{
list<int> mylist;
for (int i=1; i<=5; i++) mylist.push_back(i);
cout << "mylist contains:";
list<int>::reverse_iterator rit;
for ( rit=mylist.rbegin() ; rit != mylist.rend(); ++rit )
cout << " " << *rit;
```

```
cout << endl;
return 0;
}
```

Program wyświetli:
mylist contains: 5 4 3 2 1

Jak widać powyżej, użyto metody *push_back(i),* która dodaje element „i” na koniec listy.

Warto znać też inne metody:

Właściwości listy:
empty() - sprawdza czy lista jest pusta. Jeżeli tak zwraca wartość „true”.
size() - zwraca liczbę elementów listy
max_size() - zwraca maksymalną liczbę elementów
resize() - zmienia rozmiar listy
Dostęp do elementów listy:
front() - zwraca pierwszy element listy
back() - zwraca ostatni element listy
Modyfikacja elementów listy:
push_front() - dodaje element na początek listy
push_back – dodaje element na koniec listy

pop_front() – usuwa pierwszy element listy

pop_back() – usuwa ostatni element listy
insert() - wstawia element do listy

Skoro użyliśmy dodawania na koniec listy, dodajmy też elementy na początek list:

```
#include <iostream>
#include <list>
using namespace std;
int main ()
{
list<int> mylist (2,100); // dwa int-y z równe 100
mylist.push_front (200);
mylist.push_front (300);
cout << "mylist contains:";
for (list<int>::iterator it=mylist.begin(); it!=mylist.end(); ++it)
cout << " " << *it;
cout << endl;
return 0;

}
```

Program zwróci:
300 200 100 100

Warto również wytestować metodę insert():

```
list<int> mylist; // tworzy liste
list<int>::iterator it; // do wskazywania na elementy
// listy
for (int i=1; i<=5; i++) mylist.push_back(i); // po wykonaniu lista zawiera: 1 2 3 4 5
it = mylist.begin(); // teraz it pokazuje na poczatek listy
++it; // teraz it pokazuje na 2 element
mylist.insert (it,10); // po wykonaniu lista zawiera: 1 10 2 3 4 5
```

Warto znać również inne metody:
erase(iterator pozycja) - usuwa element listy
swap() - wymienia zawartość dwóch list
clear() - czyści zawartość listy
splice(iterator position, list<T,Allocator>& x) - łączy dwie listy
remove (const T& value) - usuwa elementy o danej wartości
sort – sortuje elementy listy
reverse – odwraca kolejność listy

Mając przetestowane podstawowe metody naszej biblioteki, możemy przejść do tworzenia programu docelowego. Zadaniem tego programu jest możliwość dodawania danych osobowych do bazy danych osób. Baza danych ograniczać będzie się tylko do dwóch danych tj. identyfikatora oraz nazwiska osoby. Dodajemy zatem strukturę:

```
struct Osoby{
        int id;
        char nazwisko[20];
};
```

Następnie w *main* dopisujemy kod odpowiedzialny za stworzenie listy o elementach typu zgodnego z naszą strukturą:

```
list<Osoby> lista_osob;
```

Oraz wskaźnik, który pozwoli dodać elemetny do listy:

```
Osoby *os=new Osoby;
```

Pamiętać należy, że wskaźnik ten należy usunąć na końcu *main'a* użuwając do tego operatora *delete*. Teraz dodajemy kolejne osoby, w sposób jak pokazano poniżej:

```
os->id=0;
strcpy(os->nazwisko,"Maklowicz");
//dodaj element na listę
lista_osob.push_back(*os);
```

Oczywiście należy dodać kilka kolejnych osób o różnych nazwiskach i id (co najmniej trzy osoby).

Ostatecznie należy również dodać funkcję wyświeltającą całą listę oraz funkcję która odszuka osobę o podanym nazwisku.
Funkcja szukacjąca powinna przyjmować argument jakim jest nazwisko oraz lista, natomiast zwracać 1 jeśli odszukano osobę oraz 0 jeśli tej osoby nie odszukano. Delarację funkcji wyświetlajacej dane oraz szukającej powinny być następujące:

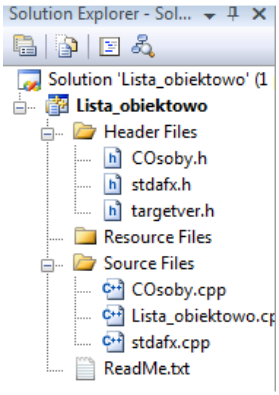
```
void wyswietl(list <Osoby> *lista);
int szukaj(char * nazwisko,list <Osoby> *lista);
```

Natomiast main naszego programu może być następujący:

```
int _tmain(int argc, _TCHAR* argv[])
{

        int w;
        char nazwa[20];
        list<Osoby> lista_osob;
        Osoby *os=new Osoby;
```

```
os->id=0;
strcpy(os->nazwisko,"Maklowicz");
//dodaj element na listę
lista_osob.push_back(*os);

os->id=1;
strcpy(os->nazwisko,"Mizerny");
//dodaj element na listę
lista_osob.push_back(*os);

os->id=2;
strcpy(os->nazwisko,"Stoklosa");
//dodaj element na listę
lista_osob.push_back(*os);

wyswietl(&lista_osob);

cout << "Nazwisko do odszukania: ";
cin>> nazwa;
if(szukaj(nazwa,&lista_osob)==1)
cout << "Osobnik wystepuje na liscie " << endl;
else cout << "Brak takiego osobnika " << endl;

system("pause");
delete os;



return 0;
}
```

Nasz program jest skoczony, jednak nie jesteśmy zadowolenia, ponieważ to co zrobiliśmy nijak wpisuje się w temat programowania zorientowanego obiektowego. Aby zmnienić wygląd naszego projektu dodamy do niego klasę. Klasa ta musi posiadać pole jakim jest lista oraz musi zawierać metody wykonujące dodawnie elemetntu do listy, wyświetlające listę oraz szukające osoby na liście. Najwygodniej jest stworzyć nowy projekt konsolowy. Widok naszego projektu powinien przypominać rysunek:



Zatem do pustego projektu dodajemy plik „COsoby.h” którego zawartość zamieszczono poniżej:

```
#pragma once
#include <string.h>
#include <list>
#include <iostream>

using namespace std;
```

```
struct Osoby{
	int id;
	char nazwisko[20];
};


class COsoby
{
	public:
	COsoby(void);
	~COsoby(void);

	public:
	int szukaj(char * nazwisko);
	void wyswietl(void);
	void dodaj_element(char *nazwisko, int id);

	public:
	list<Osoby> lista_osob;
	Osoby *os;

};
```

Teraz na podstawie powyższych deklaracji należy uzupełnić plik „COsoby.cpp”. W gotowym projekcie *main* będzie miał postać:

```
#include "stdafx.h"
#include "COsoby.h"

COsoby *Baza_danych;

int _tmain(int argc, _TCHAR* argv[])
{
	Baza_danych = new COsoby;
	Baza_danych->dodaj_element("Malajkat", 0);
	Baza_danych->dodaj_element("Sobczak", 1);
	Baza_danych->dodaj_element("Witczak", 2);
	Baza_danych->wyswietl();
	Baza_danych->szukaj("Malajkt");

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

Jak widać, dodajemy w nim bazę danych oraz trzy pozycję do bazy, następnie wyświetlamy całośc oraz sprawdzamy czy „Malajkt” występuje na liście. W wyniku wykonania programu w konsoli możemy uzyskać:

```
Liczba elementow listy:3
 0 Malajkat
 1 Sobczak
 2 Witczak
Osobnik: 'Malajkt' nie wystepuje na liscie:
Aby kontynuować, naciśnij dowolny klawisz . . .
```

Ostatnim etapem jest dodanie funkcji zaprzyjaźnionej. To czym jest przyjaźń pomiędzy funkcją a klasą było tematem jednego z wykladów z Informatyki. Nie będziemy zatem wnikać w szczegóły. Warto jednak, przyponieć najważniejsze cechy *przyjaźni*:

- jej dodanie polega na dodaniu słowa „friend” przed nazwą funkcji w klasie, z którą ma być zaprzyjaźniona,

- nie jest medotą klasy, jednak ma dostęp do wszystkich jej skladników,
- można dodać ją w każdym miejscu klasy po etykietach private, protected oraz public (etykiety nie zmnieniają jej działania),
- klasa pochodna nie dziedziczy przyjaźni,
- zaprzyjaźnić można również dwie klasy, wówczas jedna z klas ma dostęp do elementów drugiej klasy (i nie na odwrót!)

Naszym zadaniem jest dodanie funcji zaprzyjaźnionej do klasy „COsoby”. Wpisujemy zatem „przyjaciela” do naszej klasy:

```
class COsoby
{
    friend void ocena(COsoby *Baza, int id);

    public:
    COsoby(void);
    ~COsoby(void);
```

Teraz należy dodać definicję naszej funkcji w pliku COsoby.cpp:

```
void ocena(COsoby *Baza, int id)
{
    list<Osoby>::iterator it;
    int w=0;
    for (it=(Baza->lista_osob.begin()); it!=(Baza->lista_osob.end()); ++it){
            if(id==it->id)
            {

                cout << "Id: '"<<id <<"' wystepuje na liscie:"<<endl;
                cout << " " << it->id<< " " << it->nazwisko<<endl;
                w=1;
            }
        }
        if(w==0)
            cout << "Id: '"<<id <<"' nie wystepuje na liscie:"<<endl;


}
```

W *main* należy użyć naszej funkcji zaprzyjaźnionej pisząc np.:
ocena(Baza_danych, 7);

Funcja ta porówna id=7 z id elementów na liście (bazie danych). Oczywiście to zastosowanie nie jest najlepszym, ponieważ łatwiej dodać metodę która dokona porównania elementów na liście. Jest to jednak jeden z wielu przykładów na to jakie możliwości daje nam „przyjaźń” można bowiem dokonać zaprzyjaźnienia dwóch różnych klas z jedną i tą samą funkcją. Ta jednak właściwość zostanie sprawdzona jako zadanie domowe.

Sprawozdanie:

1. Do zakończonego na zajęciach projektu dodać kolejną klasę, która będzie tworzyła bazę danych. Analogicznie wykorzystać metody i pola jakich użyto w klasie *COsoby*. Nowa baza danych ma opierać się na strukturze:

```
struct Osoby{
      int numer;
     char nazwisko[20];
     char imie[20];
      int wiek;
};
```

Należy również zmodyfikować funkcję zaprzyjaźnioną *void ocena(…)* tak aby dokonywała ona przeszukiwania „starej” bazy danych oraz „nowej”, a następnie „dopisywała” brakujace elementy z „nowej” bazy danych do „starej” bazy danych. Uwaga: funkcja „void ocena(…)” musi być zaprzyjaźniona z klasami „starej” i „nowej” bazy danych. Dołączyć zmodyfikowany kod oraz napisać stosowny komentarz.