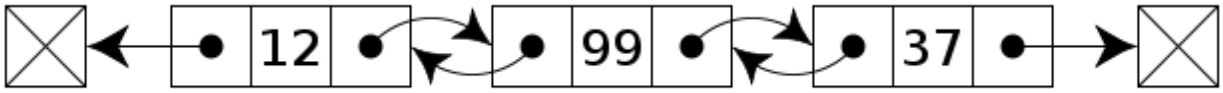
Języki programowania, wtorek 18.01.2022
Zadanie 11 – ostatnie

Tworzymy listę podwójnie powiązaną (ang. *double-linked list*).
Listy są jednymi z najważniejszych struktur danych. Wyobraźmy sobie, że tworzymy bazę danych jakiś produktów, które sprzedajemy w naszym sklepie, np. gier komputerowych. Nie wiemy, ile tych gier będzie w naszym sklepie. Co więcej, ich liczba będzie się zmieniać w zależności od tego, ile będziemy ich sprzedawać oraz ile z nich będziemy otrzymywać z dostaw. Oczywiście moglibyśmy taką bazę danych stworzyć poprzez deklarowanie dużej tablicy gier komputerowych w naszym programie. Jeśli jednak mówimy tu o wielkościach rzędu setek tysięcy, to deklarowanie „na wyrost" np. 100 tysięcy więcej miejsc w naszej bazie niż ilość elementów, powodowałoby znaczną stratę ilości pamięci. Oczywiście jest to tylko ilustracja problemu. Jednym ze sposobów, by poradzić sobie z takim problemem są tzw. dynamiczne struktury danych.
Listy są dynamicznymi strukturami danych, które mogą zmieniać swój rozmiar w trakcie działania programu. Lista będzie miała zatem taką długość i rozmiar, jaka jest w niej aktualnie liczba elementów i długość ta oraz rozmiar będą się zmieniać w zależności od dodawania bądź usuwania z niej elementów.
Budowa listy dwukierunkowej (podwójnie powiązanej) jest następująca: każdy jej element (rekord) posiada swoje **dane** oraz **2 wskaźniki** – do elementu poprzedniego oraz następnego. Powstaje zatem łańcuch powiązań elementów pomiędzy sobą, ilustruje to schemat poniżej:



A doubly linked list whose nodes contain three fields: an integer value, the link to the next node, and the link to the previous node.

Zauważamy, że lista posiada ponadto 2 specjalne, wyróżnione elementy – pierwszy i ostatni (nazywane różnie, np. head i tail, first i last, itp.), które wskazują na koniec naszego łańcucha (czyli na `NULL`). Od razu widzimy, że bezpośredni dostęp do dowolnego elementu takiej struktury jest niemożliwy (mamy bezpośredni dostęp jedynie do pola początkowego lub końcowego). Żeby więc wybrać interesujący nas element ze środka listy, musimy ustawić się na początku lub końcu listy i po niej przeiterować do wybranej pozycji.
Podczas jednego z laboratoriów używaliśmy już listy zawartej w Standardowej Bibliotece Szablonów języka C++ (tzw. STL). Dzisiaj chcemy niejako sami zaimplementować podobną listę.

Dla uproszczenia napiszemy listę zmiennych typu `double`. Implementacja listy w języku C++ polega w ogólności na stworzeniu klasy `List` oraz prostej klasy `Element`. Struktura typu `Element` powinna zawierać wartość oraz wskaźniki na element następny oraz poprzedni. Klasa `List` powinna zawierać wskaźnik na element pierwszy w liście oraz na element ostatni w liście. Plik **List.h** powinien wyglądać następująco:

```
class List
{
  struct Element
  {
    double value; //nasze dane - wartosc typu double
    Element *prev; //wskaznik na element poprzedni
    Element *next; //wskaznik na element nastepny
  };

  int fCapacity; //rozmiar listy
  Element *firstElement; //wskaznik na pierwszy element w liscie
  Element *lastElement; //wskaznik na ostatni element w liscie

 public:
  List(); //konstruktor domyslny
  List(List& list); //konstruktor kopiujacy
  ~List(); //destruktor

  void AddLast(double val); //dodaje element na koncu listy
  void AddFirst(double val); //dodaje element na poczatku listy
  void AddAt(int id, double val); //ustawia element na pozycji id
  void RemoveLast(); //usuwa element z konca listy
  void RemoveFirst(); //usuwa element z poczatku listy
  void RemoveAt(int id); //usuwa element z pozycji id
  double GetLast(); //pobiera ostatni element z listy
  double GetFirst(); //pobiera pierwszy element z listy
  double GetAt(int id); //pobiera element z pozycji id
  void Clear(); //usuwa wszystkie elementy z listy
  int Capacity(); //zwraca rozmiar
```

```
  void PrintList(); //wypisyje liste na ekran
};
```

Opis poszczególnych metod:

- Konstruktor domyślny:
  Tworzy obiekt typu `List` i ustawia pole `fCapacity` na 0 oraz wskaźniki `firstElement` i `lastElement` na `NULL` (oznacza to, że tworzymy pustą listę bez żadnego elementu).
- Konstruktor kopiujący:
  Kopiuje pola `fCapacity`, `firstElement` i `lastElement` tworząc nowy obiekt `List` z podanego obiektu `List`.
- Destruktor:
  Jeśli lista nie jest pusta, to iteruje po wszystkich elementach listy i usuwa je operatorem `delete`. Jeśli lista jest pusta, to zwraca na ekran tekst „List is empty!”.
- `void AddLast(double val)`:
  Dodaje element na końcu listy i zwiększa `fCapacity` o 1.
  Wskazówka: należy stworzyć, używając operatora `new`, obiekt typu `Element`, przyporządkować jego polu `value` podaną wartość `val`, zaś wskaźnik na element nstępny `next` ustawiamy na `NULL`. Następnie sprawdzamy, czy lista jest pusta, czy nie (czy dostawiamy pierwszy element, czy już są jakieś elementy w liście). Jeśli lista **nie jest pusta**, wskaźnik `prev` naszego nowego obiektu typu `Element` ustawiamy na `lastElement`. Na końcu musimy ustawić (przesunąć) wskaźnik `lastElement` na nasz nowy obiekt. Jeśli lista **jest pusta**, wskaźnik `prev` ustawiamy na `NULL`, zaś `lastElement` i `firstElement` na stworzony obiekt typu `Element` (mamy tylko jeden element, więc wskaźniki na pierwszy i ostatni muszą na niego wskazywać). Analogicznie realizujemy wszystkie inne metody!
- `void AddFirst(double val)`:
  Dodaje element na początku listy i zwiększa `fCapacity` o 1.
- `void AddAt(int id, double val)`:
  Dodaje element na konkretnym miejscu w liście, podanym jako parametr `id`, i zwiększa `fCapacity` o 1.
- `void RemoveLast()`:
  Usuwa ostatni element z listy i zmniejsza `fCapacity` o 1.
- `void RemoveFirst()`:
  Usuwa pierwszy element z listy i zmniejsza `fCapacity` o 1.
- `void RemoveAt(int id)`:
  Usuwa element z listy na pozycji `id` i zmniejsza `fCapacity` o 1.
- `double GetLast()`:
  Zwraca wartość `value` ostatniego elementu z listy.
- `double GetFirst()`:
  Zwraca wartość `value` pierwszego elementu z listy.
- `double GetAt(int id)`:
  Zwraca wartość `value` elementu z listy na pozycji `id`.
- `void Clear()`:

1. Czyści całą listę – działa prawie identycznie jak destruktor (używamy tego na obiekcie typu `List`, gdy nie chcemy go usunąć ostatecznie, tak jak to robimy operatorem `delete`, chcemy tylko pozbyć się wszystkich elementów wpisanych do listy, czyli usunąć je operatorem `delete` i ustawić `firstElement` oraz `lastElement` na `NULL` i `fCapacity` na 0).

- `int Capacity()`:
  Zwraca rozmiar `fCapacity` – obecny rozmiar tablicy.
- `void PrintList()`:
  Iteruje po elementach listy i wypisuje wszystkie na ekran w formacie typu:
  `id: 0, value: 5.67`
  `id: 1, value: 7.89`

Do wykonania:

1. W pierwszej kolejności w pliku **List.cpp** tworzymy konstruktor, konstruktor kopiujący, metodę `void AddLast(double val)`, `int Capacity()` oraz `void PrintList()`. W pliku **main.cpp** tworzymy obiekt typu `List` używając operatora `new`, dodajemy dwa dowolne elementy i wypisujemy listę i jej rozmiar na ekran. **1 p.**
2. Jeśli poprzedni punkt działa, dodajemy metody `void AddFirst(double val)`, void `RemoveLast()`, `void RemoveFirst()` i sprawdzamy, czy działają. **1 p.**
3. Dodajemy pozostałe metody oraz destruktor i sprawdzamy, czy działają (na końcu usuwamy całą listę operatorem `delete`). **1 p.**

4. Porównujemy działanie naszej listy i listy z biblioteki STL (`#include <list>`), tworząc lużywając operatora `new` obiekt typu `list<double>` (przykład: `list<double> *listStd = new list<double>();`). Dodajemy obu listom 2 takie same elementy na końcu oraz po 1 na początku (**Przypomnienie!** metody `push_back` i `push_front`). Potem dodajemy po jednym elemencie na pozycji nr 2 (**Uwaga!** Aby to zrobić w liście z STL musimy użyć `list<double>::iterator`, przykład:

```
list<double>::iterator iter = listaStd->begin();
iter++; //przesuwa wskaznik iter na pole o id=1
iter++; //przesuwa wskaznik iter na pole o id=2
listaStd->insert(iter,89.45); //wstawia wartosc na pole id=2
```

Wypisujemy elementy naszej listy za pomocą metody `PrintList()`, zaś listy z STL używając iteratora. **Przypomnienie:**

```
int i = 0;
for(iter=listaStd->begin();iter!=listaStd->end();iter++)
{
   cout<<"id: "<<i<<", value: "<<*iter<<endl;
   i++;
}
```

Na końcu zaś usuwamy wszystkie elementy metodą `Clear()` naszej listy i `clear()` listy z STL. **1 p.**

5. **<u>Klasa szablonowa</u> 1 p.**

Wstęp – co to są klasy i funkcje szablonowe w języku C++?

Rozważmy prostą klasę, która przechowuje jakiś typ informacji, na przykład tablicę `int`'ów:

```
class KlasaInt
{
      int *tablica
      int rozmiar;
      public:
      KlasaInt(int Rozmiar);
      void UstawElement(int poz, int wartosc);
      int PobierzElement(int poz);
};

KlasaInt::KlasaInt(int Rozmiar)
{
      rozmiar = Rozmiar;
      tablica = new int[Rozmiar];
}
void KlasaInt::UstawElement(int poz, int wartosc)
{
      tablica[poz]=wartosc;
}
int KlasaInt::PobierzElement(int poz)
{
      return tablica[poz];
}

int main()
{
      KlasaInt kl(10);
      kl.UstawElement(5) = 20;
      cout<<kl.PobierzElement(5)<<endl;


      return 0;
}
```

Co by należało zrobić, gdybyśmy potrzebowali przechować w identyczny sposób dodatkowo inne typy danych. np. `double`, `std::string`, `char*`, etc.? Musielibyśmy za każdym razem, dla każdego oddzielnego typu danych, który jest nam potrzebny, tworzyć nową klasę (np. `KlasaString`, gdzie mielibyśmy `string *tablica` zamiast `int *tablica` itp.). Napisanie sporej ilości tego typu klas zajęłoby nam pewną ilość czasu oraz zwiększyło znacznie ilość plików w projekcie (można sobie wyobrazić duży projekt, gdzie każdy dodaje klasy różniące się właściwie tylko typem przechowywanych danych a służących do tych samych celów – ogrom klas!). Na szczęście język C++ przychodzi nam tu z rozwiązaniem takiego problemu i pozwala tworzyć <u>klasy szablonowe</u> (ang. *template class*), które dostosowują się automatycznie do typu danych używanego w programie głównym. Przykład klasy szablonowej analogicznej do

przykładu powyżej jest następujący:

```
template<class T>
class KlasaSzablonowa
{
      T *tablica
      int rozmiar;
      public:
      KlasaSzablonowa(int Rozmiar);
      void UstawElement(int poz, T wartosc);
      T PobierzElement(int poz);
};

template<class T>
KlasaSzablonowa<T>::KlasaSzablonowa(int Rozmiar)
{
      rozmiar = Rozmiar;
      tablica = new T[Rozmiar];
}

template<class T>
void KlasaInt<T>::UstawElement(int poz, T wartosc)
{
      tablica[poz]=wartosc;
}

template<class T>
T KlasaInt<T>::PobierzElement(int poz)
{
      return tablica[poz];
}

int main()
{
      KlasaSzablonowa<int> klInt(10);
      klInt.UstawElement(5) = 20;
      cout<<klInt.PobierzElement(5)<<endl;

      KlasaSzablonowa<string> klString(10);
      klString.UstawElement(5) = "tekst";
      cout<<klString.PobierzElement(5)<<endl;

      return 0;
}
```

Załóżmy teraz, że w naszej klasie szablonowej mamy również metodę `Dodaj`, która zwraca sumę elementów dwóch obiektów z wybranych pól tablicy `tablica`, czyli:

```
template<class T>
T KlasaSzablonowa<T>::Dodaj(int poz1, int poz2)
{
      T obiekt = tablica[poz1]+tablica[poz2];
      return obiekt;
}
```

Taka metoda zadziała dla prostych typów danych, np. `int`, `double`, `string`, itp. (czyli takich, gdzie mamy zdefiniowany operator + i takie działanie ma sens). Co by się jednak stało, gdyby nasza klasa przechowywała na przykład napisane wcześniej obiekty klasy `Histogram`?(czyli coś bardziej skomplikowanego)? Taka metoda oczywiście nam nie zadziała dla histogramu (nie zdefiniowaliśmy przeciążonego operatora + dla klasy `Histogram` i jego użycie wyrzuciłoby nam błąd). Nic nie szkodzi! Język C++ przynosi nam i tutaj rozwiązanie – można tworzyć specjalizowane metody w klasach szablonowych, oto przykład metody `Dodaj` dla histogramów:

```
template<>
Histogram KlasaSzablonowa<Histogram>::Dodaj(int poz1, int poz2)
{
      Histogram c = tablica[poz1];
      for(int i=0;i<c->IloscBinow())
            c.UstawLiczbeZliczen(i,tablica[poz1]->PobierzLiczbeZliczen(i)
+tablica[poz]->PobierzLiczbeZliczen(i));
      return c;
      //to tylko przykład - tu robimy cokolwiek z obiektami typu Histogram, np.
tworzymy wyjsciowy histogram gdzie w kazdym binie liczba zliczen jest suma z
```

```
histogramow tablica[poz1] i tablica[poz2]
}
```
W naszej klasie możemy mieć wiele specjalizowanych metod (również wiele specjalizowanych metod o tej samej nazwie – dla różnych obiektów). Kompilator w trakcie kompilowania programu sam zdecyduje, którą metodę ma użyć (w przypadku wszystkich typów obiektów użyje metody ogólnej `Dodaj` z operatorem +, ale jeśli nasza klasa będzie akurat przechowywać histogramy, to automatycznie użyje metody specjalizowanej dla klasy `Histogram`).

Uwaga!

W przypadku klas szablonowych wszystko (zarówno definicję klasy jak i metody klasy) piszemy tylko w pliku **.h** (czyli najczęściej dla jednej klasy jeden plik .h) – nie możemy pisać definicji metod w pliku **.cpp**!

W języku C++ oprócz klas szablonowych możemy również tworzyć funkcje szablonowe. Np. załóżmy, że mamy kilka klas, które mają taką samą metodę (dajmy na to `int GetRozmiar()`). Chcielibyśmy stworzyć funkcję, która porównuje rozmiar dwóch obiektów dowolnej z tych klas i zwraca większą wartość. Nic trudnego, możemy dostawić więcej typów danych:

```
template<class T1, class T2>
int CompareRozmiar(T1 obiekt1, T2 obiekt2)
{
	if(obiekt1.GetRozmiar()>obiekt2.GetRozmiar())
		return obiekt1.GetRozmiar();
	else
		return obiekt2.GetRozmiar();
}
```

Oczywiście możemy dodawać więcej typów klas w nagłówku `template`, możemy je też dowolnie nazywać (np. `template<class A, class B, class C, class D>` itp.).

Chcemy rozszerzyć jej działanie napisanej na wcześniejszych zajęciach klasy `List` na elementy dowolnego typu.

Zasada działania naszej klasy z punktu widzenia programu będzie praktycznie identyczna jak listy ze standardowej biblioteki szablonów STL, np.:

```
List<double> *myList = new List<double>(); //tworzy liste przechowujaca typ double za pomoca naszej klasy List
list<double> *myListSTL = new list<double>(); //tworzy liste przechowujaca typ double za pomoca klasy z biblioteki STL
```

Uwaga!

Należy posiadać poprawnie działającą wersję klasy `List` oraz poprawnie działającą klasę z histogramem – `Histogram`.

Klasy szablonowe piszemy **tylko w plikach .h** – nie tworzymy pliku **.cpp** (ponieważ klasa szablonowa jest kompilowana dopiero przy jej użyciu w trakcie wykonywania programu).

`Używamy pliku Makefile w najbardziej zaawansowanej postaci.`