Języki i metody programowania 2

Lab 3

Statyczne tablice

W C++11 został wprowadzony dodatkowy kontener implementujący statyczną tablicę w pliku nagłówkowym

```
#include <array>
```

Deklaracja nowej tablicy:

```
//inicjalizacja tablicy intów o rozmiarze 3 za pomocą zadanych wartości:
array<int, 3> someNumbers {1,2,3};
//alternatywnie jednorodna inicjalizacja tablicy boolów o romiarze 10:
array<bool, 10> flags;
flags.fill(false);
```

Po tablicy można iterować:

```
for (auto tableElement: tab) {
  cout << "value: " << tableElement << endl
}
```

Zbiory

Zbiór, czyli kolekcja niepowtarzalnych elementów. Dostępna jest po dołączeniu:

```
#include <set>
```

Przykład użycia zbioru:

```
set<int> someNumbers {4,5,6,7};
if (someNumbers.find(5) != someNumbers.end()) {
    cout<<"5 jest elementem zbioru"<<endl;
}
if (someNumbers.find(17) != someNumbers.end()) {
    cout<<"17 jest elementem zbioru"<<endl;
} else {
    cout<<"17 nie jest w zbiorze";
}
```

Słowo kluczowe using

Za pomocą słowa kluczowego using w C++11 można wprowadzić kilka różnych definicji.

1. selektywny import symboli do globalnej przestrzeni nazw

```
using ::std::unique_ptr;
```

2. definicja nowego aliasu dla typu (odpowiednik **typedef**)

```
#include <string>
using Url = std::string;


bool IsValid(const Url &url);
```

3. import wszystkich symboli z danej przestrzeni nazw (niezalecane)

```
using namespace std;
```

Inteligentne wskaźniki

Inteligentne wskaźniki automatycznie zarządzają dostępem do powierzonej im pamięci i gdy pamięć nie jest już używana zostaje automatycznie zwolniona. Opisane poniżej funkcjonalności weszły w standardzie C++14.

Wskaźnik unique_ptr

Najprostszy z inteligentnych wskaźników. Jedyny właściciel pamięci, gdy zmienna traci zakres pamięć jest automatycznie zwolniona. Definicja znajduje się w pliku nagłówkowym

```
#include <memory>
```

Definiowanie wskaźnika i przydzielenie mu nowego obiektu Type:

```
unique_ptr<Type> pointerName = make_unique<Type>();
```

Zapis z wykorzystaniem auto:

```
auto pointerName = make_unique<Type>();
```

Wskaźnika `unique_ptr` można używać jak zwykłego wskaźnika:

```
unique_ptr<Type> pointerName = make_unique<Type>();
cout<< pointerName->value;
```

Unikalny wskaźnik jest jedynym właścicielem pamięci, dlatego jeśli chcemy przypisać obiekt `unique_ptr` do innego musimy jawnie przenieść element:

```
unique_ptr<int> origin = make_unique<int>(3);

//Błąd kompilacji:
unique_ptr<int> new_owner = origin;

//Przeniesienie odpowiedzialności:
unique_ptr<int> real_new_owner = move(origin);

if (oring == nullptr) {
   cout<<"W tym momencie utworzony obiekt o wartości 3 już nie należy do origin"<<endl;
}
```

Wtedy funkcja, która jako parametr przyjmuje `unique_ptr` rząda tak naprawdę przekazania zarządzania nad obiektem.

Typ pary i krotki

Typ pary i krotki stanowi w C++11 odpowiednik anonimowej struktury, którą można użyć w dowolnym miejscu kodu, bez potrzeby jej definiowania. Np.

```
std::pair<std::string, int> user {"abc",17};
std::tuple<std::string,double,std::string> userHello {"Mam",17.3,"lat"};
```

Przekazywanie argumentów do funkcji

Argumenty do funkcji można przekazywać w C++ na wiele różnych sposobów, natomiast należy brać pod uwagę czytelność zarówno kodu klienta (wywołujący metodę), jak i kod implementujący metodę. Stąd na laboratoriach będziemy korzystać z następującej konwencji:

1. argumenty o typie **Type** (np. std::string) do funkcji przekazujemy przez **const Type &** w celu uniknięcia potencjalnego kopiowania pamięci (referencja), ale zabezpieczamy się przed modyfikacją argumentu w metodzie (const)

   ```
   std::string ToString(const Type &t);
   ```

2. wyjątek stanowią argumenty typów prostych takich jak int, double, bool, itd. które przekazujemy przez wartość

   ```
   std::string ToString(bool b);
   ```

3. jeśli chcemy zwrócić z funkcji dwa lub więcej elementów można wykorzystać typy **std::tuple** i **std::pair**

   ```
   std::pair<bool, char> NextChar(char c);
   ```

4. jeśli zachodzi potrzeba zmodyfikowania argumentu wewnątrz funkcji przekazujemy argument przez wskaźnik (klasyczny)

```
void Modify(Type *t);
```

Zadania

1. Przygotuj metodę, która wyliczy różnicę czasu pomiędzy czasami w formacie HH:MM lub H:MM w minutach. Spośród wielu godzin należy znaleźć najmniejszą różnicę między dwoma godzinami.

Sygnatury metod:

```
unsigned int ToMinutes(std::string time_HH_MM);
unsigned int MinimalTimeDifference(std::vector<std::string> times);
```

Przydatne importy:

```
#include <vector>
#include <sstream>
#include <regex>
#include <cmath>
```

2. Napisz bibliotekę wspierającą budowę drzew binarnych z wykorzystaniem wskaźników inteligentnych.

   Użyj struktury danych: SmartTree

   Sygnatury metod:

```
std::unique_ptr <SmartTree> CreateLeaf(int value);
std::unique_ptr <SmartTree> InsertLeftChild(std::unique_ptr<SmartTree> tree, std::unique_ptr<SmartTree> left_subtree);
std::unique_ptr <SmartTree> InsertRightChild(std::unique_ptr<SmartTree> tree, std::unique_ptr<SmartTree> right_subtree);
void PrintTreeInOrder(const std::unique_ptr<SmartTree> &unique_ptr, std::ostream *out);
std::string DumpTree(const std::unique_ptr<SmartTree> &tree);
std::unique_ptr <SmartTree> RestoreTree(const std::string &tree);
```

Przydatne importy:

```
#include <ostream>
#include <string>
#include <memory>
```