Instytut Informatyki UMCS
Zakład Technologii Informatycznych

Wykład 9

Biblioteka standardowa, cz. 1

dr Marcin Denkowski

Lublin, 2019

AGENDA

1. Smart pointery
2. pair, tuple, ratio
3. chrono
4. random

SMART POINTERY

- Smart pointer – klasa mająca własności zbliżone do wskaźników
- Może być używany prawie wszędzie tam gdzie zwykłe wskaźniki
- Przeładowane operatory dereferencji (*) i dostępu (->)
- Umożliwiają efektywniejsze zarządzanie pamięcią

SMART POINTERY

- Biblioteka standardowa ma:
  - `unique_ptr` (od C++11) – unikalna własność obiektu
  - `shared_ptr` (od C++11) – współdzielona własność obiektu
  - `weak_ptr` (od C++11) – słaba referencja na `shared_ptr`
  - `auto_ptr` (do C++17) – całkowita własność obiektu

UNIQUE_PTR

- Zarządza posiadanym wskaźnikiem
- Zapewnia, że jest jedynym zarządcą danego obiektu
- Usuwa obiekt wskazywany wskaźnikiem
- Dokonuje transferu obiektu w przypadku kopiowania
- Zdefiniowana jest specjalizacja dla tablic
- Metody:
  - `konstruktor (+ „kopiujący”)`
  - `operator*` – dereferencja
  - `operator->` – dostęp do składowej
  - `operator=`
  - `operator bool`
  - `get` – zwraca „goły” wskaźnik
  - `release` – zwraca „goły” wskaźnik i przestaje nim zarządzać
  - `reset` – niszczy zarządzany obiekt
  - operator[] – dostęp do i-tego elementu (tylko w specjalizacji dla tablicy)

SHARED_PTR

- Zarządza posiadanym wskaźnikiem
- Zapewnia współdzielenie zasobu przez zliczanie referencji
- Usuwa obiekt wskazywany wskaźnikiem jeżeli ilość referencji == 0
- Zdefiniowana jest specjalizacja dla tablic (C++17)
- Metody:
  - `konstruktor (+ kopiujący)`
  - `operator*` – dereferencja
  - `operator->` – dostęp do składowej
  - `operator=`
  - `operator bool`
  - `get` – zwraca „goły” wskaźnik
  - `reset` – podmienia zarządzany obiekt
  - use_count – ilość współdzielonych referencji
  - operator[] – dostęp do i-tego elementu (tylko w specjalizacji dla tablicy) (C++17)
- Funkcje:
  - make_shared – konstruuje obiekt shared_ptr (nie robi podwójnej kopii, tylko domyślny deleter)
  - operatory relacji

WEAK_PTR

- Nie-zarządzająca referencja na obiekt zarządzany przez `shared_ptr`
- Aby uzyskać dostęp do obiektu musi być skonwertowany do `shared_ptr`
- Służy do śledzenia obiektów `shared_ptr` bez zwiększania licznika referencji

- Metody:
  - `konstruktor (tylko z shared_ptr)`
  - `operator=`
  - `get` – zwraca „goły” wskaźnik
  - `reset` – podmienia zarządzany obiekt
  - use_count – ilość współdzielonych referencji
  - expired – test czy obiekt został już skasowany
  - lock – tworzy shared_ptr na swojej bazie

WEAK_PTR, PRZYKŁAD

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

void observe(weak_ptr<int> weak) {
  if (auto observe = weak.lock()) {
    cout << "\tobserve() able to lock weak_ptr<>, value=" << *observe << endl;
  } else {
    cout << "\tobserve() unable to lock weak_ptr<>" << endl;
  }
}

int main() {
    weak_ptr<int> weak;
    cout << "weak_ptr<> not yet initialized" << endl;
    observe(weak);
    {
        auto shared = std::make_shared<int>(42);
        weak = shared;
        cout << "weak_ptr<> initialized with shared_ptr" << endl;
        observe(weak);
    }
    cout << "shared_ptr<> has been destructed due to scope exit" << endl;
    observe(weak);
}
```

PARA

- Para zmiennych jest reprezentowana przez klasę

```
template<class T1, class T2> struct pair
{
    typedef T1 first_type;
    typedef T1 second_type;
    T1 first;
    T2 second;
    pair(T1, T2);
    pair(const pair<T1,T2>&);
};
```

- oraz funkcje pomocnicze

```
• template <class T1, class T2>
  pair<V1,V2> make_pair (T1&& x, T2&& y);

• template <size_t I, class T1, class T2>
  typename tuple_element< I, pair<T1,T2> >::type&
  get(pair<T1,T2>&  pr)
```

TUPLA (KROTKA)

- Tupla zmiennych jest reprezentowana przez klasę

```
template<class... Types> class tuple;
```

- oraz funkcje pomocnicze

```
• template <class... Types>
  tuple<VTypes...> make_tuple (Types&&... args);

• template <size_t I, class... Types>
  typename tuple_element< I, tuple<Types...> >::type&
    get(tuple<Types...>& tpl)

• template <class... Tuples>
    tuple<CTypes...> tuple_cat (Tuples&&... tpls);

• template<class... Types>
    tuple<Types&...> tie (Types&... args) noexcept;
```

RATIO

- Klasa `ratio` reprezentuje skończone wymierne liczby zapisywane za pomocą licznika (*numerator*) i mianownika (*denominator*)

```
template<intmax_t N, intmax_t D = 1> class ratio
{
    typedef N num;
    typedef D den;
};
```

- oraz funkcje arytmetyczne

```
• ratio_add
• ratio_subtract
• ratio_multiply
• ratio_divide
```

- funkcje relacji

```
• ratio_equal
• ratio_not_equal
• ratio_less
• ...
```

RATIO

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

int main()
{
  typedef ratio<1,3> one_third;
  typedef ratio<2,5> two_fifths;

// wypisanie ulamkow
  cout << one_third::num  << "/" << one_third::den  << endl;
  cout << two_fifths::num << "/" << two_fifths::den << endl;

// suma ulamkow
  typedef ratio_add<one_third, two_fifths> sum;
  cout << "sum=" << sum::num << "/" << sum::den;
  cout << "=" << double(sum::num)/sum::den << endl;

  cout << "1 kilogram = " << std::kilo::num/std::kilo::den << " grams";

  return 0;
}
```

<CHRONO>

- Biblioteka czasu
- Wszystkie elementy w przestrzeni **`std::chrono`**
- 3 koncepty:
  - **Durations** – miara czasu w minutach, godzinach, etc.
    Klasa: **`duration`**
  - **Time points** – odnośnik do konkretnego punktu w czasie

    Klasa: **`time_point`**
  - **Clocks** – połączenie time point z fizycznym czasem

    Klasa: **`system_clock, steady_clock, high_resolution_clock`**

DURATION

```
template <class Representation, class Period = ratio<1> >
class duration;
```

- **`Representation`** – typ arytmetyczny, wewnętrzna reprezentacja ilości
- **`Period`** – ułamek reprezentujący okres czasu w sekundach
- Zdefiniowane `typedefs`:

typ	Representation	Period
`hours`	uint 23bit	`std::ratio<3600,1>`
`minutes`	uint 29bit	`std::ratio<60,1>`
`seconds`	uint 35bit	`std::ratio<1,1>`
`milliseconds`	uint 45bit	`std::ratio<1,1000>`
`microseconds`	uint 55bit	`std::ratio<1,1000000>`
`nanoseconds`	uint 64bit	`std::ratio<1,1000000000>`

DURATION

```
#include <iostream>
#include <ratio>
#include <chrono>

int main(){
  std::chrono::duration<int> pr1; //<int, ratio<1,1>>
  std::chrono::seconds pr2(10);

                    // counts: pr1 pr2
                    //          --- ---
  pr1 = pr2;                 // 10  10
  pr1 = pr1 + pr2;           // 20  10
  ++pr1;                     // 21  10
  --pr2;                     // 21   9
  pr1 *= 2;                  // 42   9
  pr1 /= 3;                  // 14   9
  pr2 += ( pr1 % pr2 );      // 14  14

  std::cout << "pr1==pr2: " << (pr1==pr2) << std::endl;
  std::cout << "pr1: " << pr1.count() << std::endl;
  std::cout << "pr2: " << pr2.count() << std::endl;

  return 0;
}
```

TIME POINTS

```
template <class Clock, class Duration=typename Clock::duration>
  class time_point;
```

- Klasa `time_point` wyraża punkt w czasie relatywnie do początku epoki
- Wewnętrznie zawiera obiekt `duration`
- Metoda:

  ```
  duration time_since_epoch() const;
  ```

  zwraca czas od początku epoki (zwykle 01.01.1970 00:00:00)

OPERACJE NA TIME_POINT I DURATION

Dla `time_point` i `duration` jest szereg przeciążonych operatorów arytmetycznych (i odpowiednio z minusem)

- `time_point::operator+=( const duration& d );`
- `operator+(duration, duration)`
- `operator*(duration, duration)`
- `operator/(duration, duration)`
- `operator%(duration, duration)`
- `operator+(time_point, duration)`
- `operator+(time_point, time_point)`

umożliwiające operacje arytmetyczne na „czasie”

CLOCKS

```
class system_clock;
class steady_clock;
class high_resolution_clock;
```

- Klasy zapewniające dostęp do aktualnego `time_point`
- Wszystkie mają metodę

```
static std::chrono::time_point<std::chrono::system_clock> now()
```

- Klasa `system_clock` ma ponadto statyczne metody konwersji do/z `time_t`:

```
static std::time_t
system_clock::to_time_t( const time_point& t )

static std::chrono::system_clock::time_point
system_clock::from_time_t( std::time_t t )
```

KONWERSJA TIME_POINT NA STRING

```
system_clock::time_point now = system_clock::now();

// time_t z biblioteki ctime z libc
std::time_t tt = system_clock::to_time_t(now);
std::cout << "today is: " << ctime(&tt);


// lub z formatowaniem

std::tm * ttm = localtime(&tt);
char date_time_format[] = "%Y-%m-%d %H:%M:%S";
char time_str[32] = {0};
strftime(time_str, 32, date_time_format, ttm);
std::cout << time_str << endl;
```

POMIAR CZASU

```
#include <iostream>
#include <ratio>
#include <chrono>
using namespace std::chrono;

int main ()
{
  steady_clock::time_point t1 = steady_clock::now();

      // cos dlugotrwalego

  steady_clock::time_point t2 = steady_clock::now();

  duration<double> diff = duration_cast<duration<double>>(t2 – t1);
//lub
  microseconds diff = duration_cast<microseconds>(t2 - t1);

  std::cout << "Czas: " << diff.count() << " s" << std::endl;

  return 0;
}
```

RANDOM

- Biblioteka generacji liczb pseudolosowych
- Składa się z dwóch elementów

  1) **Generatory** – obiekty które generują liczby całkowite z rozkładem jednorodnym (*Uniform Random Bit Generators URBG*)

  2) **Dystrybucje** – obiekty, które przekształcają sekwencje liczb wygenerowanych przez generator (*URBG*) w sekwencje spełniającą dany rozkład

- Przykład:

```
std::default_random_engine generator;
std::uniform_int_distribution<int> distrib(1,6);
int dice_roll = distrib(generator);
```

GENERATORY

- Generatory liczb pseudolosowych
  - `default_random_engine`
  - `mnistd_rand`
  - `mnistd_rand`
  - `mt19937`
  - `mt19937_64`
  - `ranlux24_base`
  - `ranlux48_base`
  - `ranlux24`
  - `ranlux48`
  - `knuth_b`
- Generator liczb losowych
  - `random_device`
- Metody:
  - `min()` – zwraca najmniejszą możliwą wartość
  - `max()` – zwraca największą możliwą wartość
  - `seed(result_type)` – reinicjalizacja stanu
  - **`operator()()` – zwraca nową wartość (psudo)losową**
  - `discard(long z)` – opuszcza z (pseudo)losowych wartości

DYSTRYBUCJE

- Dystrybucje losowe wykonują post-processing wyjścia generatorów tworząc liczby zgodne z danym rozkładem prawdopodobieństwa
  - `uniform_int_distribution`
  - `uniform_real_distribution`
  - `normal_distribution`
  - `lognormal_distribution`
  - `chi_squared_distribution`
  - `student_t_distribution`
  - `poisson_distribution`
  - `exponential_distribution`
  - `gamma_distribution`
  - `bernoulli_distribution`
  - `discrete_distribution`
- Metody:
  - `konstruktor` – parametry zależne od klasy
  - reset() – resetuje wewnętrzny stan dystrybucji
  - `min()` – zwraca najmniejszą możliwą wartość
  - `max()` – zwraca największą możliwą wartość
  - **`operator()(URBG)` – zwraca nową wartość (psudo)losową**
  - `param()` – akcesor parametrów rozkładu

DYSTRYBUCJA NORMALNA, PRZYKŁAD

```
std::default_random_engine generator;
std::normal_distribution<float> normdist(5, 1);

std::cout << "min=" << normdist.min() << " max="
          << normdist.max() << std::endl;

const int N = 16768;
const int BINS = 10;

// generowanie tablicy liczb o rozkładzie Gaussa
float narray[N];
for(int i=0;i<N;i++) narray[i] = normdist(generator);

// tworzenie histogramu wylosowanych liczb
int narrn[BINS] = {0};
for(int i=0;i<N;i++) narrn[(int)narray[i]]++;

for(int i=0;i<BINS; i++) cout << i << ": " << narrn[i] << endl;
```