C++11

Rdzeń języka I ulepszenie funkcjonalności

4 kwietnia 2016

Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie
AGH University of Science and Technology

Agenda

- Nowy for
- Automatyczna dedukcja typu
- Nowe typy
- Jednorodne sposoby inicjalizacji
- Nowe słowa kluczowe do użytku wraz z klasami I ich metodami
- Liczby losowe
- Wyrażenia regularne
- Czas

For (range-based)

Tablica :

```
int tab[] = { 1, 3 , 5, 7};
for(int element : tab)
{
	std::cout << element << " "; // 1 3 5 7
}
```

Vector :

```
std::vector<char> myVector = { 'T', 'e', 's',
't'};
for(char c : myVector)
{
	std::cout << c; //Test
}
```

Własna implementacja

Metody :

- const_iterator begin() const
- const_iterator end() const

```
class Backpack{

    std::vector<Item> items;
public:
    void add(const Item item)
    {
        items.push_back(item);
    }

    typedef typename std::vector<Item>::const_iterator const_iterator;

    const_iterator begin() const {return items.begin();}
    const_iterator end() const {return items.end();}
};
```

decltype

- Zwraca typ wyrażenia które zostanie mu podane w nawiasach

```
int i = 33;
decltype(i) j = i * 2;
```

```
std::vector<int> myVector = { 1, 3 , 5, 7 };
for( decltype(myVector.size()) i = 0; i < myVector.size(); ++i)
        std::cout << myVector.at(i) << " ";
```

auto

- Automatycznie dedukuje typ zmiennej z jej inicjalizacji

```
auto item = Item(10, "item");
```

- Potrafi automatycznie określić typ zwracanej zmiennej w funkcji

```
auto sum(int a, int b) -> int
{
    return a + b;
}
```

```
template <typename T>
auto mutliply(T first, T second) -> decltype( first * second )
{
    return first * second;
}
```

nullptr

- Pokazuje na zerowy(nullowy) obszar pamięci
- Dobry jako wartość początkowa wskaźników które nie zostały zaalokowane lub wartość końcowa po dealokacji zajmowanego przez nie miejsca

❑ Która funkcja się wykona?

```
void fun(char *);
```

```
void fun(int);
```

```
fun(NULL);
fun(nullptr);
```

Typy standardowe

- Jak **duży** jest char?
  - ✓ Jest wielkości 1 bajta. Musi mieć minimum 8 bitów.
- Jak **duże** są inne typy?

sizeof(`char`) <= sizeof(`short`) <= sizeof(`int`) <= sizeof(`long`)

- C++11
  - `int8_t`
  - `uint8_t`
  - `int16_t`
  - `uint16_t`
  - `int32_t`
  - `uint32_t`
  - `int64_t`
  - `uint64_t`

Uniform Initalization

- Initialization List

```
std::vector<int> intVector = {1, 3, 5, 7, 9};
std::set<int> intSet = {1, 2 ,3 , 1};
std::map<int,std::string> intKeyBasedMap { {0,"zero"}, {1,"one"}, {2,"two"} };
```

- Wspieranie list inicjalizacyjnych we własnych klasach
  - std::initializer_list<T> jako parametr konstruktora
  - Posiada : begin(), end() , size()

Spójrzmy na 8.own_list_initialization.

Uniform Initalization

```
// błąd przed C++11
NotAggregate variable = {10, 40};
// błąd przed C++11
NotAggregate secondVariable{20, 25};

Aggregate thirdVariable{5, 10};
```

```
NotAggregate createDefaultNotAggregate()
{
 //zamiast return NotAggregate(20, 50)
    return {20, 50};
}
```

```
//zamiast normalize(Vector3(2,5,9));
Vector3 x = normalize({2,5,9});
```

```
class NotAggregate{
private :
    int var1;
    int var2;
public :
NotAggregate(int var1,
int var2)
: var1(var1), var2(var2)
    {}};
```

```
struct Aggregate
{
   int var1;
   int var2;
};
```

Uniform Initalization

- Pułapki

```
int variable = 5.2; // warning
int secondVariable = {5.2}; // Nie skompiluje się
```

```
std::vector<int> firstVector{1,2,3,4,5}; //tablica 5-elementowa
std::vector<int> secondVector{10}; //tablica 1-elementowa
std::vector<int> thirdVector(10); //tablica 10-elementowa

vector<int> v = { 1, 4.3, 4, 0.6 }; //Error


Bar bar(Foo()); //Deklaracja funkcji!
Bar bar{Foo{}}; //Inicjalizacja zmiennej!

int n;

auto y {n}; // std::initializer_list<int>
auto z = {n}; // std::initializer_list<int>
```

Final

```
class Base
{
  virtual void virtualFunction() final {}
};

class Derived final : public Base
{
    //error
    virtual void virtualFunction(){ }
    void foo() final {} //error
};

class ShallWe : public Derived //error
{

};
```

- Final nie pozwala dziedziczyć po klasie
- Final nie pozwala na nadpisanie (override) metody
- Final może być użyty jako nazwa zmiennej

Override

```
struct Base
{
    virtual void foo();
    virtual void fun();
    void bar();
};

//override jako nazwa klasy!
struct override : public Base
{
    void foo() const override; // błąd!
Sygnatura funkcji się nie zgadza!
    void foo() override; // OK
    void bar() override; // błąd! bar
nie jest wirtualny
    void fun() final override; // OK
};
```

- Override zapewnia nas o tym że nadpisujemy funkcje wirtualną
- Strzeże nas przed błędami
- Wykorzystywany jest tylko tuż po deklaracji funkcji.

Delete

```
struct something
{
    void* operator new(std::size_t) =
delete;
};
something *wsk = new something; //ERROR!

struct myType
{
    myType(const myType&) = delete;
    myType() {}
};
myType t2(t1); //Error

struct Bar
{
    void fun();
};
Bar::fun() = delete; //Tak nie można
```

- Pozwala na usunięcie funkcji które są nam niepotrzebne.
- Każda próba ich wywołania to błąd kompilacji

Default

```
struct Bar
{
    Bar() = default;
    static void fun()
    {
        Bar test;
        Bar test2(test); // działa!
    }
private :
    Bar(const Bar&) = default;

};

Bar test;
Bar test2(test); //nie działa
```

- Pozwala nam na ponowne zdefiniowanie domyślnych funkcji (takich jak konstruktory)
- Można zmienić sposób dostępu do takich funkcji

Delegating constructors

```
class Complex
{
public :
    Complex(double re, double im) :re(re), im(im) {}
    Complex(const Complex& other) : Complex(other.re, other.im) {}
/* .... */
};
```

- Pozwala na wywołanie innego konstruktor podczas konstrukcji obiektu.
- Upraszcza kod.

Noexcept

```
template <typename T>
auto myMultiply(T t1, T t2)
noexcept(noexcept(t1 * t2))
-> decltype(t1 * t2)
{
    return t1 * t2;
}



PlusNumber test{2};
cout << noexcept(myMultiply(1, 1));//1
cout<<noexcept(myMultiply(test, test))//0
```

- Jest specyfikatorem który określa czy funkcja rzuca wyjątek, czy też nie.
- A co gdy taka funkcja rzuca wyjątek?

Static_assert

```
struct Teen
{
    void run() noexcept
    { std::cout << "I am walking“;}
};
struct Old
{
    void run() noexcept(false)
    { throw "ouch!"; }
};

template <class Human>
void safeRun(Human human)
{
    static_assert(noexcept(human.run()),
"This human cannot run!");

    human.run();
}
```

- Składnia :

Static_assert (bool_expr, message)
Message – pomijalny od **C++17**

- Jeśli wyrażenie jest nieprawdziwe to rzuca błąd kompilacji

Explicit operator

```
class Complex
{
public :
    explicit operator double() const
    {
        return sqrt(pow(re,2) + pow(im,2));
    }
    /*...*/
};
Cout << "Modul liczby zespolonej : " << myComplex// blad
Cout << "Modul liczby zespolonej : " << static_cast<double>(myComplex) << endl;
```

- Podobny do konstruktora explicit znanego z C++03
- Nie pozwala na niejawne wykorzystanie operatora konwersji

Liczby losowe

- Problem liczb losowych w C++03.
- Rand(), co robi źle?
  - Zwraca liczbe z przedziału [0, RAND_MAX]
  - X = rand() % Range;
  - Co gdy RANGE > RAND_MAX+1?
- Czas na rozwiązanie w C++11

Liczby losowe

- Generatory – służą do generacji liczb

Generator	Opis
linear_congruential_engine	Podobny do rand()
mersenne_twister_engine	Przedział - $[0,2^w-1]$
subtract_with_carry_engine	Algorytm typu “Lagged Fibonaccci”
random_device	Używa hardware’u

- Adaptory – Zmieniają działanie generatorów

Adaptor	Opis
discard_block_engine	Odrzuca pewną część liczb
independent_bits_engine	Określa długość bitową liczb
shuffle_order_engine	Zmienia kolejność występowania

Liczby losowe

- Instancje

Instancja	Opis
default_random_engine	Prosty w użyciu
minstd_rand0	linear_congruential_engine
minstd_rand	linear_congruential_engine
mt19937	mersenne_twister_engine
mt19937_64	mersenne_twister_engine
ranlux24_base	subtract_with_carry_engine
ranlux48_base	subtract_with_carry_engine
ranlux24	discard_block_engine
ranlux48	discard_block_engine
knuth_b	shuffle_order_engine

Liczby losowe

- Rozkłady – Zmieniają sekwencje liczb tak aby zachowywały właściwości danego rozkładu .
  - Jednorodny
  - Bernoulliego
  - Normalne
  - Poissona
  - Bazujące na prawdopodobieństwie/częstotliwości

seed_seq

O sposobie na generacje ziarna

- Pozwala na użycie więcej niż jeden zmiennej na inicjalizacje ziarna
- Możliwośc generacji dużej ilości ziarn z jednego źródła zmiennych
- Tworzy ziarna o wartości od 0 do 2^{32}.
  Dlatego `uint32_t` nadaje się znakomicie do ich obsługi

Zobaczmy kod.

Ratio

```
typedef std::ratio<1, 3> jednaTrzecia;
typedef std::ratio<1, 2> jednaDruga;

display<jednaTrzecia>(); // 1/3
std::cout<< " + "; display<jednaDruga>();
// 1/2
std::cout << " = ";

typedef std::ratio_add<jednaDruga,
jednaTrzecia> sum;

display<sum>(); // 5/6
```

- Reprezentuje ułamki
- Oblicza wszystko podczas kompilacji

- Ułamki możemy mnożyć, dodawać, dzielić I porównywać.

Ratio

- Niektóre wbudowane ułamki

type	definition	description
micro	ratio<1,1000000>	10^{-6}
milli	ratio<1,1000>	10^{-3}
centi	ratio<1,100>	10^{-2}
deci	ratio<1,10>	10^{-1}
deca	ratio<10,1>	10^{1}
hecto	ratio<100,1>	10^{2}
kilo	ratio<1000,1>	10^{3}
mega	ratio<1000000,1>	10^{6}
giga	ratio<1000000000,1>	10^{9}

Chrono

```
using std::chrono::system_clock;

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

std::time_t tt;

tt = system_clock::to_time_t ( today );
std::cout << "today is: " << ctime(&tt);
```

- Służy do operacji związanych z czasem.
- Składa się z :
  - Duration
  - Time Point
  - Clock

Chrono::duration

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

- Reprezentuje przedział czasowy
- Period jest wyrażony w sekundach
- Zdefiniowane szablony :
  - hours
  - minutes
  - seconds
  - miliseconds
  - nanoseconds

Chrono clocks

- cystem_clock
  Zwraca czas systemowy.

- steady_clock
  Najlepszy do zwracania czasu który posłuży do mierzenia czasu
  Jaki upłynął między dwoma zdarzeniami

- high_resolution_clock
  Mierzy czas z dużą dokładnością

Regex

Obsługa wyrażeń regularnych

- Funkcje :
  - regex_match
  - regex_search
  - regex_replace
- Obiekty
  - basic_regex (regex)
  - match_results (smatch)
  - sub_match

Regex

Przydatne flagi

- Dla basic_regex
  - icase
  - optimize
- Dla funkcji
  - match_continuous
  - format_first_only
  - match_not_null
  - format_no_copy

Bibliografia

- http://stackoverflow.com/
- http://www.stroustrup.com/
- http://en.cppreference.com/
- http://www.cplusplus.com/
- http://programmers.stackexchange.com/
- https://mbevin.wordpress.com/2012/11/16/uniform-initialization/