



C++ 11:

Pętla `for` oparta na zakresie

43

C++11: pętla `for` oparta na zakresie

Nowa składnia pętli `for`

```
for ( deklaracja : wyrażenie ) instrukcje
```

wyrażenie – tablica/kontener/dowolny obiekt implementujący semantykę kontenera (zwraca obiekt w stylu iteratora za pomocą metod `begin` i `end`)

deklaracja zmiennej - typu takiego jak elementy w tablicy lub kontenerze

instrukcje – wykonywane tyle razy, ile elementów przechowuje tablica lub kontener

Przykład:

```
vector<int> vec;
vec.push_back( 10 );
vec.push_back( 20 );

for (int i : vec) {
    cout << i;
}
```

© UKSW, WMP, SNS, Warszawa

44

C++11: pętla `for` oparta na zakresie

Przykłady

Modyfikowanie zawartości kontenera:

```
vector<int> vec;
vec.push_back( 1 );
vec.push_back( 2 );

for (int& i : vec)
{
    i++; // inkrementuje wartości z wektora
}
```

© UKSW, WMP, SNS, Warszawa

45

C++11: pętla `for` oparta na zakresie

Obiekty posiadające zakres to:

- wszelkie kontenery z biblioteki STL, ponieważ:
 - posiadają iteratory, które wspierają metody `operator*`, `operator!=` i `operator++` (zadeklarowane jako metody własne albo jako funkcje globalne),
 - posiadają metody `begin` i `end` (zadeklarowane jako metody własne albo jako funkcje globalne), zwracające iteratory na początek i koniec kontenera.
- wszelkie inne obiekty zaimplementowane samemu o funkcjonalności kontenerów STL (obowiązkowa w/w funkcjonalność).

© UKSW, WMP, SNS, Warszawa

46



C++ 11:

Dedukcja typów danych

47

C++11: dedukcja typów danych

Mechanizmy dedukcji typów danych

Pochodzący C++98:

- Szablony i ich parametry

Wprowadzone w C++11:

- `auto`
- `decltype`

- Pozwalają uniknąć ciągłego wypisywania typów, które są oczywiste bądź powtarzają się.
- Zmiana typu zmiennej w jednym miejscu kodu propaguje się na cały kod (kompilator sam dedukuje nowy typ danych – *uwaga: może nas zaskoczyć*).

© UKSW, WMP, SNS, Warszawa

48

C++11: dedukcja typów danych

Zasady dedukcji dla **auto**

Podobne do zasad dedukcji dla parametrów szablonu z C++98

Deklarując **auto** jako typ zmiennej

```
auto x = 10;
// auto odgrywa rolę taką, jak parametr T w szablonie:
template <typename T>
void fun(T x) ;
fun(10);
```

Stąd:

```
const char name[] = "ABC"; // name: const char[13]
auto arr1 = name;          // arr1: const char*

void Fun(int, double);      // Fun: void(int, double)
auto func1 = Fun;           // func1: void (*)(int, double)
```

© UKSW, WMP, SNS, Warszawa

49

49

C++11: dedukcja typów danych

Pożytek z **auto**

Mamy szablon funkcji, która na pewnym zakresie elementów kontenera identyfikowanym przez parę iteratorów, wykonuje „czynności ważne i potrzebne”:

```
template<typename It> // wykonuje czynności ważne i potrzebne
void czwip(It b, It e) // na rzecz elementów od 'b' do 'e'
{
    for (; b != e; ++b) {
        typename std::iterator_traits<It>::value_type currValue = *b;
        ...
    }
}
```

Odwolujemy się pracowicie do cech charakterystycznych iteratora, żeby pozyskać typ danych wskazywanych przez te iteratory.

© UKSW, WMP, SNS, Warszawa

50

50

C++11: dedukcja typów danych

Pożytek z **auto**

Mamy szablon funkcji, która na pewnym zakresie elementów kontenera identyfikowanym przez parę iteratorów, wykonuje „czynności ważne i potrzebne”:

```
template<typename It> // wykonuje czynności ważne i potrzebne
void czwip(It b, It e) // na rzecz elementów od 'b' do 'e'
{
    for (; b != e; ++b) {
        auto currValue = *b;
        ...
    }
}
```

Wersja dla mniej pracowitych. ☺

© UKSW, WMP, SNS, Warszawa

51

51

C++11: dedukcja typów danych

Pożytek z **auto**

Jeszcze jeden przykład dla „mniej pracowitych”:

```
map<string, string> address_book;
for ( auto address_entry : address_book )
{
    cout << address_entry.first << " < " <<
        address_entry.second << ">" << endl;
}
```

© UKSW, WMP, SNS, Warszawa

52

52

C++11: dedukcja typów danych

Pożytek z **auto**

Kiedy bezpieczniej jest pisać **auto** – przykład: pozyskanie rozmiaru kontenera:

```
std::vector<int> v;
...
unsigned sz = v.size();
```

Co prawda typ zwracany przez `.size()` to `vector<int>::size_type` ale wszyscy wiedzą, że to typ całkowity bez znaku, więc spotyka się **unsigned**.

Nie wszyscy jednak wiedzą, że choć w Win32 zmienne tych dwóch typów zajmują tę samą liczbę bajtów, to już w Win64 nie (**unsigned** zajmuje mniej) ..

Rekompilacja kodu na Win64 powiedzie się, ale dla dostatecznie dużych zbiorów danych działanie może przestać być poprawne. ☹

Dlatego prościej i bezpieczniej pisać: `auto sz = v.size();`

© UKSW, WMP, SNS, Warszawa

53

53

C++11: dedukcja typów danych

Niespodzianki z **auto**

Przykład: mamy funkcję, która dla pewnej klasy widgetów zwraca wektor opisujący ich cechy, tj. co potrafią, a czego nie:

```
std::vector<bool> cechy(const Widget& w);
```

Kod programu:

```
Widget w;
...
bool cecha5 = cechy(w)[5]; // czy posiada cechę nr 5
..
```

A gdyby tak napisać:

```
auto cecha5 = cechy(w)[5]; // czy posiada cechę nr 5
```

Powinno działać tak samo.

© UKSW, WMP, SNS, Warszawa

54

54

C++11: dedukcja typów danych

Niespodzianki z `auto`

Dla:

```
auto cecha5 = cechy(w)[5];
```

Typ zmiennej `cecha5` to `vector<bool>::reference`
(klasa zagnieżdżona w `vector<bool>`).

Powód jest natury technicznej: klasa `vector<bool>` została tak zaprojektowana, aby przechowywać wartości w postaci spakowanej – jeden bit na każdy element (a w C++ nie ma referencji do pojedynczych bitów).

Stąd operator `[]` dla `vector<bool>` zwraca obiekt, który *działa tak jak* `bool&`. Oczywiście, wśród jego wielu funkcjonalności, istnieje też możliwość konwersji tego obiektu do `bool`.

© UKSW, WMP, SNS, Warszawa

55

55

C++11: dedukcja typów danych

Niespodzianki z `auto`

Dla:

```
auto cecha5 = cechy(w)[5];
```

Typ zmiennej `cecha5` to `vector<bool>::reference`
(klasa zagnieżdżona w `vector<bool>`).

Teraz zmienna `cecha5` jest kopią obiektu przechowującego referencję do właściwej informacji – bitu nr 5.

Niestety, funkcja `cechy` zwraca referencję do obiektu, który za chwilę zniknie (funkcja `cechy` zwraca obiekt tymczasowy (kontener), więc jego elementy też są tymczasowe...).

Dlatego `cecha5` będzie przechowywał referencję do obiektu, który nie istnieje. ☹

© UKSW, WMP, SNS, Warszawa

56

56

C++11: dedukcja typów danych

Niespodzianki z `auto`

Klasa:

```
vector<bool>::reference
```

to tzw. klasa proxy (*proxy class*). Takich klas jest wiele.

Przeostrożność: należy unikać sytuacji w kodzie, gdzie:

```
auto pewnazmienna = wyrażenie typu 'klasa proxy';
```

No tak, tylko że klasy proxy zostały tak napisane, żeby być jak najmniej widoczne (w założeniu: *mają być całkiem niewidoczne*).

Aby się nie dać oszukać, należy:

sprawdzać nagłówki funkcji i metod (np. w źródłach bibliotek), albo ..

```
auto cecha5 = static_cast<bool>(cechy(w)[5]);
```

© UKSW, WMP, SNS, Warszawa

57

57

C++11: dedukcja typów danych

Dedukcja typu przez `decltype`

zwraca typ zmiennej lub wyrażenia podanego w argumentach:

```
const int i = 0;           // decltype(i): const int
bool f(const Widget& w);    // decltype(w): const Widget&
                           // decltype(f): bool(const Widget&)
```

```
struct Point {
    int x, y;               // decltype(Point::x): int
};                           // decltype(Point::y): int
```

```
Widget w;                   // decltype(w): Widget
```

```
if (f(w)) ...               // decltype(f(w)): bool
```

© UKSW, WMP, SNS, Warszawa

58

58

C++11: dedukcja typów danych

Dedukcja typu przez `decltype`

Przydaje się szczególnie kiedy szablon funkcji ma zwrócić typ danych zależny od typu parametru (ale nie będący typem parametru)

Przykład: funkcja zwracająca element z kontenera po uprzedniej autoryzacji dostępu do danych dla bieżącego użytkownika:

```
template<typename Container, typename Index>
... (?) authAndAccess(Container& c, Index i) {
```

...

Funkcja powinna zwracać typ taki, jak ma `c[i]`

Jak go pozyskać?

© UKSW, WMP, SNS, Warszawa

59

59

C++11: dedukcja typów danych

Dedukcja typu przez `decltype`

Dygresja:

W C++11 wprowadzono możliwość podania typu zwracanego przez funkcję w miejscu tuż za jej nagłówkiem (tzw. *trailing return type*); bardzo przydatne, kiedy typ zwracanych danych ma skomplikowany zapis, np.:

```
auto fpif(int) -> int(*) (int)
```

Koniec dygresji.

Drugie zastosowanie: kiedy zwracany typ zależy od argumentów wywołania (*trt*):

```
template<typename Container, typename Index>
auto authAndAccess(Container& c, Index i) -> decltype(c[i])
{
    authenticateUser();      // autoryzacja dostępu
    return c[i];             // dostęp do danych
}
```

© UKSW, WMP, SNS, Warszawa

60

60



C++ 11:

Sprytne wskaźniki (smart pointers) inteligentne wskaźniki

61

C++11: sprytne wskaźniki

Kontrola zmiennych dynamicznych

C++11 dostarcza parę szablonów klas:

1. **shared_ptr**
2. **weak_ptr**

Obiekty tych typów reprezentują wskaźniki do zmiennych dynamicznych.

Ich zadanie:

kontrolować zaalokowane zasoby i udostępniać tak długo, póki są potrzebne, ale nie dłużej. Automatycznie zwolnić każdy dynamicznie zaalokowany zasób, kiedy kasowany jest ostatni wskaźnik, który by na niego wskazywał.

To pomaga zapobiegać wyciekaniu pamięci (memory leaks).

© UKSW, WMP, SNS, Warszawa

62

C++11: sprytne wskaźniki

Kontrola zmiennych dynamicznych

Obiekty typu **shared_ptr<Ty>**

- Przechowuje wskaźnik do dynamicznie zaalokowanego zasobu typu **Ty**.
- Jeżeli wskaźnik do zasobu jest różny od NULL, do zasobu mamy dostęp operatorami: **operator->** oraz **operator***.
- Obiekt typu **shared_ptr<Ty>** przechowuje też wskaźnik do obiektu z informacją o liczbie odwołań do zasobu, na który wskazuje, tj. o liczbie obiektów typu **shared_ptr<Ty>**, które kontrolują ten sam dynamicznie zaalokowany zasób.

Jest to realizacja semantyki dzielonej własności (shared ownership semantics), gdzie każdy dynamicznie zaalokowany zasób jest własnością wskaźników, które na niego wskazują.

© UKSW, WMP, SNS, Warszawa

63

C++11: sprytne wskaźniki

Kontrola zmiennych dynamicznych

Obiekty typu **shared_ptr<Ty>**

- Kiedy tworzymy kopię obiektu typu **shared_ptr<Ty>**, licznik odwołań do tego zasobu jest inkrementowany.
- Kiedy usuwamy kopię obiektu typu **shared_ptr<Ty>**, licznik odwołań do tego zasobu jest dekrementowany.
- Kiedy licznik odwołań dla zasobu schodzi do zera, to znaczy, że żaden obiekt typu **shared_ptr<Ty>** nie wskazuje na dynamicznie zaalokowany zasób. Wtedy kończy się jego czas życia i zasób jest automatycznie zwalniany.

Realizacja od strony technicznej: Każdemu dynamicznie zaalokowanemu zasobowi przypisany jest jego własny obiekt z licznikiem odwołań i innymi informacjami. Sprytny wskaźnik kontroluje zasób oraz jego obiekt z licznikiem.

© UKSW, WMP, SNS, Warszawa

64

C++11: sprytne wskaźniki

Kontrola zmiennych dynamicznych

Obiekt typu **weak_ptr<Ty>**

- Przechowuje wskaźnik do dynamicznie zaalokowanego zasobu typu **Ty**.
- Jeżeli wskaźnik jest różny od NULL, do zasobu mamy dostęp operatorami: **operator->** oraz **operator***.
- Obiekt typu **weak_ptr<Ty>** nie kontroluje licznika odwołań do zasobu, na który wskazuje. *Mimo to, też się bardzo przydaje.*

© UKSW, WMP, SNS, Warszawa

65

C++11: sprytne wskaźniki

Kontrola zmiennych dynamicznych

shared_ptr posiada dynamicznie zaalokowany zasób, jeżeli:

1. Został utworzony z argumentem zawierającym wskaźnik do zasobu,
2. Został utworzony jako kopia obiektu posiadającego ten zasób,
3. Został utworzony z obiektu **weak_ptr**, który wskazywał na ten zasób,
4. Posiadanie zostało mu przypisane w wyniku użycia **operator=** lub **reset**. W tym przypadku przestaje posiadać poprzedni zasób, o ile taki posiadał.

weak_ptr wskazuje na zasób, jeżeli:

1. Został utworzony z obiektu **shared_ptr**, który posiadał ten zasób,
2. Został utworzony z obiektu **weak_ptr**, który wskazywał na ten zasób,
3. Wskazanie zostało mu przypisane w wyniku użycia **operator=**. W tym przypadku przestaje wskazywać na poprzedni zasób, o ile wskazywał.

© UKSW, WMP, SNS, Warszawa

66

C++11: sprytnie wskaźniki

Kontrola zmiennych dynamicznych

Obiektowi typu `shared_ptr` można przypisać kontrolę nad dynamicznie zaalokowanym zasobem lub uczynić pustym.

Obiekt typu `shared_ptr` traci kontrolę nad zasobem, kiedy przypisywana mu jest kontrola nad innym zasobem, albo kiedy jest usuwany.

Kiedy jeden lub kilka obiektów typu `shared_ptr` współposiada pewien zasób i zostaną usunięte aż do ostatniego, dynamicznie zaalokowany zasób jest automatycznie zwalniany. Jeżeli się tak stanie, wszystkie obiekty typu `weak_ptr`, które wskazywały na ten zasób, tracą ważność (*expire*).

© UKSW, WMP, SNS, Warszawa

67

67

C++11: sprytnie wskaźniki

Kontrola zmiennych dynamicznych – przykłady:

Deklaracja obiektu reprezentującego sprytny wskaźnik:

```
#include <memory>
```

```
struct resource { // przykładowa struktura do alokacji
    resource(int i0 = 0) : i(i0) {}
    int i;
};

int main() {
    shared_ptr<int> sp1; // konstruktor domyślny daje pusty obiekt
    shared_ptr<resource> sp2(new resource(3));
}
```

© UKSW, WMP, SNS, Warszawa

68

68

C++11: sprytnie wskaźniki

Kontrola zmiennych dynamicznych – przykłady:

Kontrolowany zasób ma swój usuwacz (*has a deleter*) jeżeli jego oryginalny właściciel otrzymał ten usuwacz w argumentach wywołania konstruktora, tj. obiekt typu `shared_ptr` został utworzony z dwoma argumentami konstruktora: (1) ze wskaźnikiem na zasób oraz (2) na obiekt usuwacza.

```
struct deleter {
    void operator()(resource *res) {
        cout << "destroying resource at " << (void*)res << '\n';
        delete res;
    }
};

int main() {
    shared_ptr<resource> sp(new resource(3), deleter());
}
```

Usuwanie jest przechowywane w obiekcie z licznikiem.

© UKSW, WMP, SNS, Warszawa

69

69

C++11: sprytnie wskaźniki

Kontrola zmiennych dynamicznych – przykłady:

Kopiowanie:

Uwaga: nie należy używać tego samego zasobu przy tworzeniu dwóch różnych sprytnych wskaźników, bo destruktor też będzie użyty dwa razy – należy raczej:

- pierwszy utworzyć j.w.,
- drugi – za pomocą konstruktora kopiującego z tego pierwszego.

```
shared_ptr<resource> sp1(new resource(4));
// sp1 przechowuje wskaźnik na zasób
shared_ptr<resource> sp2(sp1); // sp2 udostępnia ten sam zasób
```

© UKSW, WMP, SNS, Warszawa

70

70

C++11: sprytnie wskaźniki

Kontrola zmiennych dynamicznych – przykłady:

Dostęp do zasobu:

```
int *ip = new int(3); // alokowanie zasobu typu int
cout << (void*)ip << '\n'; // pokazanie adresu
shared_ptr<int> sp(ip); // tworzenie obiektu shared_ptr
cout << (void*)sp.get() << '\n'; // pokazanie przechowywanego adresu

cout << *sp << '\n'; // pokazanie przechowywanej wartości
cout << (void*)&*sp << '\n'; // pokazanie adresu wartości
```

© UKSW, WMP, SNS, Warszawa

71

71

C++11: sprytnie wskaźniki

Kontrola zmiennych dynamicznych – przykłady:

Dostęp do pól zasobu typu strukturalnego:

```
struct S {
    int member;
};

int main() {
    S *s = new S; // tworzenie obiektu typu S
    s->member = 4; // przypisanie do member
    shared_ptr<S> sp(s); // tworzenie obiektu shared_ptr
    if (sp) // konwersja na bool (przydatna!)
        cout << sp->member << '\n'; // pokazanie wartości member
}
```

Dostęp jest identyczny, jak w przypadku zwykłych zmiennych wskaźnikowych

© UKSW, WMP, SNS, Warszawa

72

72

C++11: sprytnie wskaźniki

Kontrola zmiennych dynamicznych – przykłady:

Liczenie sprytnych wskaźników odnoszących się do jednego zasobu:

```
typedef shared_ptr<int> spi;
spi sp0; // pusty obiekt
cout << "empty object: " << sp0.use_count() << '\n'; // 0
spi sp1((int*)0); // żadnego zasobu, ale niepusty
cout << "null pointer: " << sp1.use_count() << '\n'; // 1
spi sp2(new int); // posiada zasób
cout << sp2.unique() << '\n'; // zwraca true - jedyny właściciel
cout << "one object: " << sp2.use_count() << '\n'; // 1
{ // tworzymy krótko żyjący obiekt sp3
    spi sp3(sp2); // kopiowanie
    cout << "two objects: " << sp2.use_count() << '\n'; // 2
    cout << sp2.unique() << '\n'; // false - dwóch właścicieli
} // sp3 usunięty
cout << "one object: " << sp2.use_count() << '\n'; // 1
```

73

C++11: sprytnie wskaźniki

Kontrola zmiennych dynamicznych – przykłady:

Przekazywanie i zwalnianie zasobów:

```
sps sp0(new resource(1)); // alokacja i przypisanie zasobu
sps sp1(new resource(2)); // alokacja i przypisanie zasobu
sps sp2(sp0); // przypisanie zasobu
sp1 = sp0; // przypisanie zasobu i zwolnienie zasobu
sp2.reset(); // zwolnienie zasobu
sp0.swap(sp2); // zamiana przypisań zasobów
if(sp0 == sp1) cout << "Nie jest dobrze."
if(sp0 != sp1) cout << "Jest dobrze."
```

© UKSW, WMP, SNS, Warszawa

74

C++11: sprytnie wskaźniki

Kontrola zmiennych dynamicznych – przykłady:

Operator porównania „mniejszy-niż” i zastosowanie z kontenerem:

```
#include <algorithm> static void lookup(const iset& data, spi sp)
#include <memory> { // szuka obiektu pasującego do sp
#include <iostream> {
#include <set> {
    citer res = lower_bound(data.begin(), data.end(), sp);
using namespace std;
    cout << *sp;
    if (res == data.end() || *res != sp)
        cout << "nie ma.\n";
    else
        cout << "jest.\n";
};
typedef shared_ptr<int> spi;
typedef set<spi> iset;
typedef iset::const_iterator citer;
```

© UKSW, WMP, SNS, Warszawa

75

C++11: sprytnie wskaźniki

Kontrola zmiennych dynamicznych – przykłady:

Operator porównania „mniejszy-niż” i zastosowanie z kontenerem:

```
iset data; // kontener (pusty)
spi sp0(new int(0));
spi sp1(new int(1));
spi sp2(new int(2));
spi sp3(sp1); // dzielenie własności z sp1
spi sp4(new int(1)); // taka sama wartość jak sp1, ale inny zasób
data.insert(sp0);
data.insert(sp1);
data.insert(sp2);
lookup(data, sp1); // szukamy sp1 - sukces
lookup(data, sp3); // szukamy sp3 - sukces
lookup(data, sp4); // szukamy sp4 - ...?
```

© UKSW, WMP, SNS, Warszawa

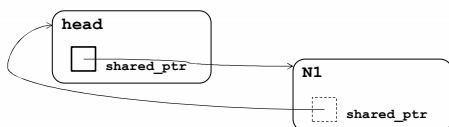
76

C++11: sprytnie wskaźniki

Kontrola zmiennych dynamicznych – **weak_ptr**:

Obiekty tego typu przydatne są do **przerywania cykli** w strukturach danych. Cykl występuje wtedy, gdy dwa lub więcej kontrolowanych zasobów przechowują wskaźniki na siebie nawzajem, tak że tworzą one pętlę.

Np. jeżeli dynamicznie zaalokowany zasób **head** zawiera pole typu **shared_ptr**, który wskazuje na inny zasób **N1**, który zawiera pole **shared_ptr**, wskazujące na **head**, to takie dwa zasoby tworzą pętlę.



© UKSW, WMP, SNS, Warszawa

77

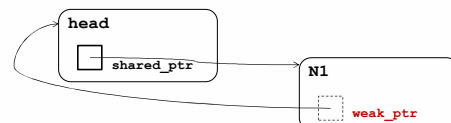
C++11: sprytnie wskaźniki

Kontrola zmiennych dynamicznych – **weak_ptr**:

Ponieważ takie dwa zasoby wskazują na siebie nawzajem, żaden nie będzie miał nigdy licznika wskazań o wartości zero.

Dlatego nigdy nie zostaną automatycznie usunięte, nawet, jeżeli – poza nimi samymi – żaden inny wskaźnik nie będzie już na nie wskazywał.

Aby to zmienić **N1** powinien zawierać pole typu **weak_ptr**, wskazujące na **head**.



© UKSW, WMP, SNS, Warszawa

78

C++11: sprytnie wskaźniki

Kontrola zmiennych dynamicznych – `weak_ptr`:

Jeżeli `N1` zawiera pole typu `weak_ptr`, wskazujące na `head`, to:

1. Nadal można sięgnąć do `head` z `N1`.
2. Kiedy ostatni `shared_ptr` wskazujący na `head` zostanie usunięty, licznik wskazań na `head` będzie równy zero i `head` zostanie usunięty.
3. Dobrze napisany destruktor `head` usunie również `shared_ptr` wskazujący na `N1` – wtedy licznik wskazań na `N1` będzie równy zero i `N1` też zostanie usunięty.

Przykład:

```
struct node { // struct do demonstracji cykli
    shared_ptr<node> next;
    weak_ptr<node> weak_next;
};
```

© UKSW, WMP, SNS, Warszawa

79

79

C++11: sprytnie wskaźniki

Kontrola zmiennych dynamicznych – `weak_ptr`:

Przykład – struktury danych z cyklem i bez cyklu:

```
node *head = new node;
node *N1 = new node;
shared_ptr<node> root(head);

head->next = shared_ptr<node>(N1); // tworzymy pierwsze powiązanie
N1->next = root; // zapętlamy wskazania - jest cykl!
N1->weak_next = root; // zapętlamy wskazania - nie ma cyklu!
```

© UKSW, WMP, SNS, Warszawa

80

80

C++11: sprytnie wskaźniki

Kontrola zmiennych dynamicznych – `weak_ptr`:

Przykład utraty ważności wskaźnika:

```
cout << boolalpha;
shared_ptr<resource> sp(new resource);
weak_ptr<resource> wp(sp);
cout << "wskazuje na zasób: " << wp.expired() << '\n';
cout << "liczba posiadaczy zasobu: " << wp.use_count() << '\n';

sp.reset(); // ustawiamy wskazanie obiektu na NULL
            // - wskazywany zasób dynamiczny traci jednego z
            // właścicieli. Tak się składa, że ostatniego (!)

cout << "stracił ważność: " << wp.expired() << '\n';
```

© UKSW, WMP, SNS, Warszawa

81

81

C++ 11: nullptr



82

C++11: nullptr

Zerowanie wskaźnika

Do wyzerowania wskaźnika posługujemy się `NULL` bądź `0`. To są liczby.

Stąd może pojawić się niejednoznaczność:

```
void fun(int); // trzy przeciążenia fun
void fun(bool);
void fun(void*);

fun(0); // wywoła fun(int), nie fun(void*)
fun(NULL); // wywoła fun(int), nigdy fun(void*)
```

Dlatego w C++11 wprowadzony został `nullptr`.

© UKSW, WMP, SNS, Warszawa

83

83

C++11: nullptr

Zerowanie wskaźnika

`nullptr` nie jest typem całkowitoliczbowym.

Działa, jakby był wskaźnikiem na każdy dowolny typ (taka magia..)

```
fun(nullptr); // wywoła fun(void*)
```

Pomaga ujednolicić kod, np.:

```
auto result = findRecord( /* argumenty */ );
if (result == 0) { // result - może wskaźnik, może int..
if (result == nullptr) {
    ...
}
```

Jakiego typu będzie `result`?

© UKSW, WMP, SNS, Warszawa

84

84

C++11: nullptr

Przykład:

```
class Widget {  
    ...  
};  
  
int f1(Widget* pw) { return 0; };  
double f2(Widget* pw) { return 0.0; };  
bool f3(Widget* pw) { return true; };  
  
template<typename FuncType,  
        typename PtrType>  
auto Call(FuncType func, PtrType ptr) -> decltype(func(ptr)) {  
    return func(ptr);  
};
```

© UKSW, WMP, SNS, Warszawa

85

85

C++11: nullptr

Przykład:

```
auto result1 = Call(f1, 0);  
// 'int (Widget *)' : cannot convert parameter 1 from 'int' to 'Widget *'  
  
auto result2 = Call(f2, NULL);  
// 'double (Widget *)' : cannot convert parameter 1 from 'int' to 'Widget *',  
  
auto result3 = Call(f3, nullptr);  
// działa!
```

© UKSW, WMP, SNS, Warszawa

86

86