C++ 11:

Sprytne wskaźniki (smart pointers)
inteligentne wskaźniki

61

C++11: sprytne wskaźniki

Kontrola zmiennych dynamicznych

C++11 dostarcza pary 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

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 zalokowany 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

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

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

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

66

C++11: sprytne 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*).

C++11: sprytne 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));
```


C++11: sprytne 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());
```

Usuwacz jest przechowywany w obiekcie z licznikiem.

C++11: sprytne 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
```


C++11: sprytne 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
```


C++11: sprytne 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

C++11: sprytne 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
```

© UKSW, WMP. SNS, Warszawa

C++11: sprytne 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

C++11: sprytne wskaźniki

Kontrola zmiennych dynamicznych – przykłady:

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

```
#include <algorithm>
#include <memory>
#include <iostream>
#include <set>

using namespace std;

typedef shared_ptr<int> spi;
typedef set<spi> iset;
typedef iset::const_iterator citer;
```

```
static void lookup(const iset& data,
                   spi sp)
{ // szuka obiektu pasującego do sp
  citer res =
   lower_bound(data.begin(),
               data.end(), sp);
  cout << *sp;
  if (res == data.end() ||
                  *res != sp)
    cout << " nie ma.\n";
  else
    cout << " jest.\n";
};
```

© UKSW, WMP. SNS, Warszawa

C++11: sprytne 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

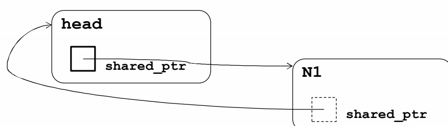
C++11: sprytne wskaźniki

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

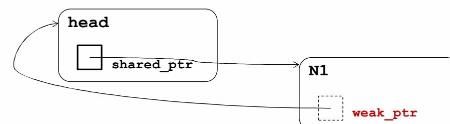
C++11: sprytne wskaźniki

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

C++11: sprytne 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

C++11: sprytne 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!
```

(Linia `N1->next = root;` jest na slajdzie przekreślona.)

© UKSW, WMP. SNS, Warszawa

C++11: sprytne 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

C++ 11:
nullptr

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

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 ujednoznacznić kod, np.:

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

(Linia `if (result == 0) {` jest na slajdzie przekreślona.)

Jakiego typu będzie **result**?

© UKSW, WMP. SNS, Warszawa

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

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