

(Średnio) zaawansowane programowanie w C++ (ZPR)

Wykład 3 - sprytnie wskaźniki

Robert Nowak

23Z

Deklaracja i definicja

Deklaracja - sygnalizacja że pewna nazwa odnosi się do „czegoś”

```
extern int x; //deklaracja obiektu
long silnia(long n); //deklaracja funkcji
class Logger; //deklaracja klasy
template<typename T> class Node; //deklaracja szablonu
```

Definicja - podanie szczegółów;

```
int x = 4; //definicja i inicjacja zmiennej x
long silnia(long n) { //definicja funkcji
    if(n < 2) return 1;
    return n * silnia(n-1);
};
```

Inicjacja - nadanie początkowej wartości

Definicja obiektu: wtedy gdy można zainicjować

Obiekt. Czas życia obiektu.

Obiekt: ma typ oraz unikalny identyfikator.

1-wartość: odnosi się do „czegoś” co istnieje w pamięci.

- ▶ automatyczne
 - ▶ tworzone w chwili napotkania definicji
 - ▶ są niszczone w momencie wyjścia z zasięgu (np. z bloku)
- ▶ statyczne (globalne i zadeklarowane jako `static`)
 - ▶ tworzone tylko raz
 - ▶ są niszczone po zakończeniu programu
- ▶ dynamiczne (tworzone na stacku)
 - ▶ operatory `new` i `delete` (uwaga! oddzielna wersja dla tablic),
 - ▶ programista steruje czasem życia obiektów (tworzy je i usuwa),
 - ▶ Nie używamy funkcji z `<stdlib.h>`, czyli `malloc`, `free` itp.
- ▶ tymczasowe

Obiekty. Konwersja typów

- ▶ **Nie używa się rzutowania w stylu 'C'!**
- ▶ nie używa się void*
- ▶ konstruktory explicit

Operatory rzutowania (unikać):

- ▶ `static_cast` - konwersję pomiędzy „spokrewnionymi” typami
- ▶ `dynamic_cast` - rzutowanie w dół hierarchii klas
- ▶ `const_cast` - znosi kwalifikator `const`
- ▶ `reinterpret_cast` - dowolna konwersja (tak jak w C)

```
double d = 3.2;  
int i = static_cast<int>(d); //przykład rzutowania
```

Mechanizmy obsługi błędów (problem komunikacji)

- ▶ autor biblioteki może wykryć błąd, nie wie, co z nim zrobić
- ▶ użytkownik wie co zrobić z błędem, nie potrafi go wykryć

- ▶ Nieprawidłowe:
 - ▶ ignorowanie błędów
 - ▶ kończenie działania programu
 - ▶ komunikaty dla użytkownika.

- ▶ Kłopotliwe:
 - ▶ kod powrotu
 - ▶ zmienna globalna
 - ▶ specjalny błędny stan obiektu
 - ▶ ANSI: longjmp

- ▶ Mechanizm wyjątków.

Mechanizmy:

kod powrotu

```
class Wektor {  
    /* ... */  
    bool add(int i); //Zwraca false gdy niepowodzenie  
};  
//Przykład użycia  
Foo foo;  
if(!foo.add(4)) //Sprawdzanie, czy nie wystąpił błąd  
    //Tutaj kod obsługi błędu
```

- ▶ rezerwacja zwracanej wartości na kod błędu
- ▶ wartość zwracana musi być badana (użytkownik może zignorować fakt wystąpienia błędu)
- ▶ kod obsługi błędów wymieszany z innym kodem

zmienna globalna

Sygnalizacja wystąpienia błędu przez ustawianie wartości zmiennej globalnej. (np. `#include <errno.h>`)

```
extern int errno; //Zmienna globalna, przechowuje kody błędów
class Plik {
    /* ... */
    void zapisz(const char* buf); //Ustawia errno gdy wystąpi błąd
};
Plik p; //Przykład użycia
p.zapisz(buforA);
if(errno == ENOFILE) //Sprawdzanie, czy wystąpił określony błąd
    //Obsługa błędu
```

- ▶ obiekt globalny - potencjalne źródło kłopotów
- ▶ należy co jakiś czas sprawdzać wartość (użytkownik może zignorować fakt wystąpienia błędu)
- ▶ kod obsługi błędów wymieszany z innym kodem

skoki (#include <csetjmp>)

```
jmp_buf buf; //struktura, która zapamiętuje stan procesora
int main(){
    if( ! setjmp(buf) ){ //rej. stan procesora. Zwraca 0
        /* tutaj obliczenia, które mogą być błędne */
    }
    else { /* wygenerowano błąd */}
}

int f(){
    longjmp(buf, 23); //skok do stanu, który przechowuje buf
    //skutkuje powrotem z funkcji setjmp z daną wartością
}
```

- ▶ identyfikacja błędów przez numer (globalnie)
- ▶ nie przekazuje dodatkowych informacji o błędach (można wspomóc się zmiennymi globalnymi)
- ▶ nie usuwa automatycznych obiektów przy wychodzeniu z bloku (nie woła destruktorów)

Mechanizm wyjątków

```
class Wektor {  
    int* tab;  
    int rozm;  
public:  
    class Zakres{ }; //Klasa wyjątku  
    int& operator[](int idx) {  
        if(idx >= 0 && idx < rozm) return tab[i];  
        else throw Zakres(); //Funkcja biblioteczna rzuca wyjątek  
    }  
};  
  
void f(Wektor& w) {  
    try { //Funkcja użytkownika będzie przechwytywać wyjątki  
        g(w);  
    }  
    catch(Wektor::Zakres) { //Procedura obsługi wyjątku  
        //kod obsługi błędu  
    }  
}
```

Wyjątki - właściwości

- ▶ nie rezerwuje wartości zwracanej (konstruktory mogą zwracać błędy)
- ▶ brak obiektów globalnych
- ▶ użytkownik nie może zignorować zgłoszonego błędu
- ▶ kod obsługi błędów oddzielony od innego kodu

Wymaga wsparcia przez język

- ▶ mechanizm „rzucania” wyjątku
 - ▶ inny mechanizm powrotu
 - ▶ korzysta ze zwijania stosu
- ▶ specyfikacja interfejsu
- ▶ niewyłapane wyjątki

Klasy, które reprezentują błędy - specjalny typ klas

- ▶ Powinny dziedziczyć po **std::exception**
- ▶ Konstruktor wyjątku nie może zgłaszać wyjątków!
- ▶ Często implementują wzorec wizytatora
- ▶ Tworzone podczas zgłaszania wyjątku w specjalnym miejscu

//Przykładowa klasa błędu przy konwersji napisu na liczbę

```
class InputException : public std::exception {  
    char bad_char;  
public:  
    InputException(char c) : bad_char(c) {}  
    InputException(const InputException& e) throw()  
        : bad_char(e.bad_char){}  
    char getBadChar() const { return bad_char;}  
};
```

Mechanizm wyjątków : koszty

- ▶ rejestracja bloków try - catch
- ▶ koszt rzucenia wyjątku wysoki, w porównaniu z kosztem zwracania obiektu, wolniej $\approx 100\times$!
 - ▶ obiekt zawsze jest kopiowany (niezależnie od tego, czy jest łapany przez referencję, wskaźnik czy wartość)
 - ▶ jeżeli obiekt jest łapany przez wartość podwójna kopia
 - ▶ nie ma możliwości optymalizacji
 - ▶ rozpoznawanie typu (w catch) wykorzystuje mechanizmy RTTI, koszt taki jak `dynamic_cast`

Wyjątki : zasady stosowania

- ▶ Hierarchie klas: możliwość grupowania błędów
- ▶ Nie należy rzucać wyjątków w destruktorach
- ▶ Wyjątek rzucać przez wartość

```
throw Exception //zgłasza wyjątek przez wartość  
throw new Exception; //zajmuje pamięć na stercie!
```

- ▶ Wyjątek przechwytywać przez referencję

```
catch(Exception& e) //przechwytuje przez referencję  
catch(Exception e) //tworzy lokalną kopię
```

- ▶ Należy reagować na wszystkie wyjątki

Kod bezpieczny przy uwzględnieniu wyjątków

Przykład:

```
class Bitmap { /* ... */ };  
class Okno {  
public:  
    /* ... */  
private:  
    Bitmap* b;  
    /* ... */  
};
```

```
Okno& operator=(const Okno& o) {  
    delete b;  
    b = new Bitmap(*o.b);  
    return *this;  
}  
  
Okno& operator=(const Okno& o) {  
    if(this == &o) return *this;  
    delete b;  
    b = new Bitmap(*o.b);  
    return *this;  
}  
  
Okno& operator=(const Okno& o) {  
    Bitmap* old = b;  
    b = new Bitmap(*o.b);  
    delete old;  
    return *this;  
}
```

testowanie funkcji uwzględniając wyjątki

```
template<typename T> void std::swap(T& a, T& b) {  
    T tmp = a;  
    a = b;  
    b = tmp;  
}
```

Testy:

//test dla typu
//wbudowanego

```
int a = 2, b = 3;  
swap(a,b);
```

//test dla typu
//użytkownika

```
Foo f1(2), f2(3);  
swap(f1 ,f2);
```

```
try { //wyjątek w operatorze przypisania  
    swap(f1, f2); //test 2  
} catch(std::exception&) {}
```

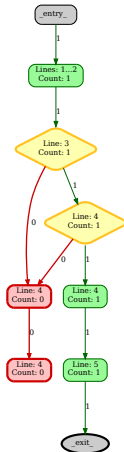
```
try { //j.w., wyjątek za drugim razem  
    swap(f1, f2); //test 3  
} catch(std::exception&) {}
```

```
try { //wyjątek w konstruktorze kopiującym  
    swap(f1, f2); //test 4  
} catch(std::exception&) {}
```

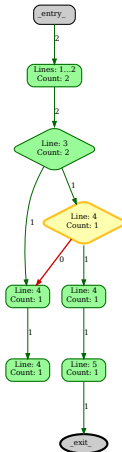
testy swap<int>
Pokrycie: 100%



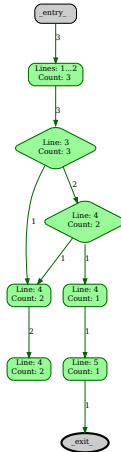
testy swap<Foo>
test 1
Pokrycie: 50%



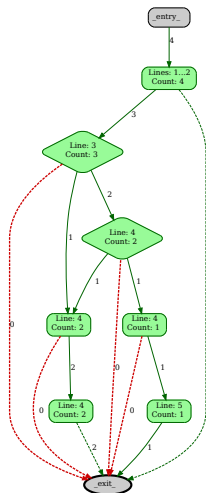
testy swap<Foo>
test1 oraz test2
Pokrycie: 75%



testy swap<Foo>
test 1, test 2, test 3
Pokrycie: 100%



testy swap<Foo>
wszystkie testy
Pokrycie: 100%



mechanizm wyjątków a zasoby

```
void f1() {  
    int* tmp = new int;  
    //Kod, który używa tmp i może rzucić wyjątkiem  
    delete tmp;  
}  
  
//Nieudolna poprawa  
void f2() {  
    int* tmp = new int;  
    try {  
        //Kod, który używa tmp i może rzucić wyjątkiem  
    }  
    catch(...) { //Przechwytuje wszystkie wyjątki  
        delete tmp; throw;  
    }  
    delete tmp;  
}
```

boost::scoped_ptr

- ▶ wzorzec RAII, „zdobywanie zasobów jest inicjowaniem”
- ▶ destruktor usuwa wskaźnik
- ▶ zabronione kopiowanie

```
template<typename T> class scoped_ptr : noncopyable {
public:
    explicit scoped_ptr(T* p = 0) : p_(p) {}
    ~scoped_ptr(){ delete p_; } //Usuwa obiekt wskazywany
    T& operator*() { return *p_; }
    T* operator->() { return p_; }
private:
    T* p_; //Wskaźnik, którym zarządza
};

void f() {
    scoped_ptr<MojaKlasa> klasa(new MojaKlasa);
    klasa->get(); //używamy klasy tak jak wskaźnika
    //kod, który może wyrzucać wyjątki
} //Destruktor klasy scoped_ptr wywoła operator delete
```

std::auto_ptr - kopiowanie jest przekazywaniem własności

- ▶ Wykorzystuje RAII
- ▶ Destruktor usuwa wskaźnik
- ▶ Dozwolone kopiowanie (!) - **przenosi ono uprawnienia**

```
template<typename T> class auto_ptr {  
public:  
    explicit auto_ptr(T* p = 0) : p_(p) {}  
    //nie jest const auto_ptr& !  
    auto_ptr(auto_ptr& a) p_(a.p_) { a.p_ = nullptr; }  
    auto_ptr& operator=(auto_ptr& a); //nie jest const auto_ptr&  
    ~auto_ptr() { delete p_; } //usuwa obiekt wskazywany  
    T& operator*() { return *p_; }  
    T* operator->() const { return p_; }  
private:  
    T* p_;  
};
```

std::auto_ptr - wycofywany, zastąpił go std::unique_ptr

C++11, r-value reference, typename &&

```
struct Foo { //do demonstracji r-value
    Foo(int i) : i_(i) {}
    int i_;
};

void f(Foo& x) { cout << "l-value" << endl; }
void f(Foo&& x){ cout << "r-value" << endl; }

int main() {
    X x(2);
    f(x); //l-value
    f(3); //r-value
}

//Możliwość definiowania konstruktorów, które przenoszą zawartość
Foo(const Foo& f); //obiekt f nie będzie zmieniany
Foo(Foo&& f); //obiekt f może być zmieniony
```

r-value reference wykorzystane w `std::unique_ptr`

std::unique_ptr

- ▶ obiekty `unique_ptr` mają wielkość zwykłego wskaźnika
- ▶ RAII, automatycznie usuwają wskazywany obiekt w destruktorze

```
//zwraca wskaźnik na obiekt
```

```
std::unique_ptr<Foo> createFoo(int n) {  
    return std::unique_ptr<Foo>(new Foo(n) );  
}
```

```
createFoo(2); //nie wykorzystana wartość zwracana zostanie usunięta
```

```
std::make_unique<Foo>(2); //to samo
```

```
{  
    std::unique_ptr<Foo> v = createFoo(3);
```

```
    //używa v
```

```
} //destruktor zwalnia zasob
```

std::unique_ptr (2)

std::unique_ptr – zawsze jest dokładnie jeden właściciel danych
przekazanie własności - konstruktor przenoszący lub funkcja move

```
std::unique_ptr<Foo> p = std::make_unique<Foo>(42);  
std::unique_ptr<Foo> q = std::move(p);  
//teraz p nie wskazuje na nic  
q.reset(); //zwolnienie obiektu, jeżeli wskaźnik był właścicielem  
q.reset(new Foo(43) ); //zwolnienie obiektu i zarządzanie nowym  
Foo* raw = q.get(); //dostęp do gołego wskaźnika, utrzymując własność
```

unique_ptr mogą być przechowywane w kontenerach standardowych

```
std::unique_ptr<Foo> p = std::make_unique<Foo>(42); //to samo  
std::vector< std::unique_ptr<Foo> > v;  
v.push_back(p); //błąd kompilacji - nie ma konstruktora kopiującego  
v.push_back( std::move(p) ); //v staje się wł. wskazywanego obiektu  
//teraz p wskazuje na null
```

Rule of three, rule of five (C++11)

Dla klasy z niebanalnymi składowymi należy dostarczyć:
destruktor, konstruktor kopiujący, operator przypisania

```
class Foo {  
    /* ... */  
    Foo(const Foo& f) : ptr_(new Goo(*f.ptr_)) {}  
    Foo& operator=(const Foo& f) { /* ... */ }  
    ~Foo() { delete ptr_; }  
private:  
    Goo* ptr_;  
};
```

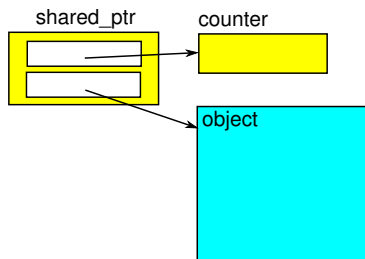
C++11: konstruktor przenoszący i przen. operator przypisania

```
/* jak wyżej */  
Foo(Foo&& f) : ptr_(f.ptr_) { f.ptr_ = nullptr; }  
Foo& operator=(Foo&& f) { /* ... */ }
```

std::unique_ptr - zasoby i wyjątki

```
using S = const string&;  
class Book {  
    string n_; Image* i_; Audio* a_;  
public:  
    //wersja 1  
    Book(S n, S in, S an): n_(n),i_(new Image(in)),a_(new Audio(an)){}  
    //wersja 2  
    Book(S n, S in, S an) : n_(n) {  
        i_ = new Image(in);  
        try { a_ = new Audio(an);} catch(...){ delete i_; throw;}  
    }  
}  
//wersja 3  
class Book {  
    string n_; unique_ptr<Image> i_; unique_ptr<Audio> a_;  
public:  
    Book(S n, S in, S an): n_(n),i_(new Image(in)),a_(new Audio(an)){}  
    ~Book() {}  
};
```


std::shared_ptr, boost::shared_ptr - sprytny wskaźnik



- ▶ konstruktor : tworzy licznik i inicjuje go na 1
- ▶ konstruktor kopiujący: zwiększa licznik odniesień
- ▶ destruktor: zmniejsza licznik odniesień, jeżeli ma on wartość 0 to kasuje obiekt

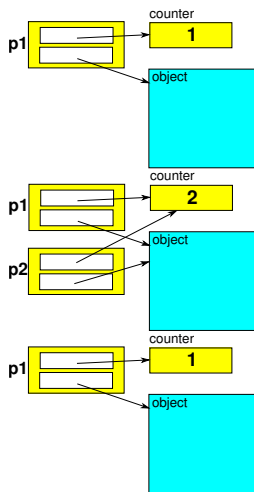
sprytny wskaźnik - przykład

```

#include <memory>
class Foo { /* ... */ };

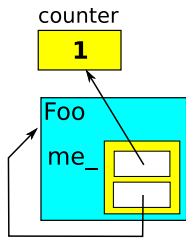
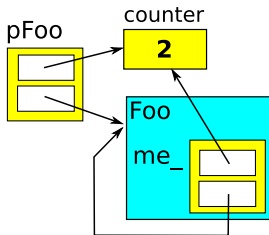
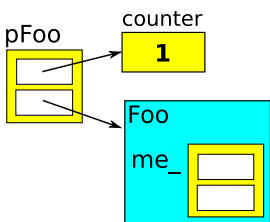
{
    std::shared_ptr<Foo> p1(new Foo(1) );
    {
        std::shared_ptr<Foo> p2(p1);
        //licznik odniesien == 2
        /* ... */
    } //destruktor p2, licznik = 1
} //destruktor p1 usuwa obiekt

```

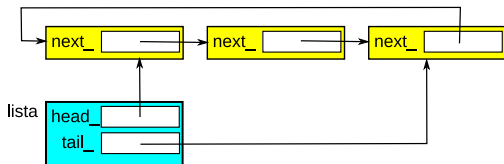


std::shared_ptr - problemy w zależnościach cyklicznych

```
struct Foo {  
    std::shared_ptr<Foo> me_;  
    /* ... */  
};  
  
std::shared_ptr<Foo> pFoo = new Foo;  
pFoo->me_ = pFoo;
```



Jawne przerywanie zależności cyklicznej



```

class List { //lista cykliczna, patrz rysunek
    using PNode = std::shared_ptr<Node>;
    struct Node { //węzeł dla listy
        Node(PNode next) : next_(next) { }
        PNode next_; //wskaźnik na element następny
    };
public:
    ~List() { if( tail_ ) tail_->next_.reset(); } //jawnie przerywa zależność
    //destruktor obiektów head_ oraz tail_ skasują elementy listy
private:
    PNode head_, tail_;
};
  
```

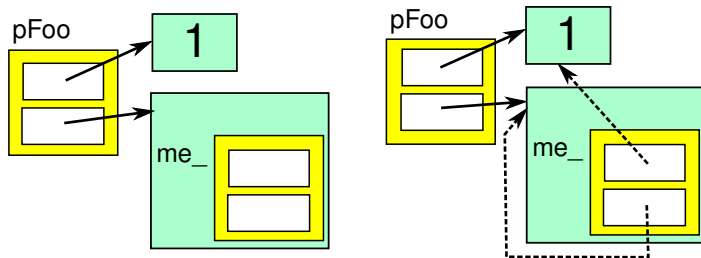
std::weak_ptr, boost::weak_ptr

Pozwala prawidłowo implementować cykliczne zależności, nie zmienia licznika odniesień

```
template<class T> class weak_ptr {
public:
    weak_ptr();
    template<class Y> weak_ptr(std::shared_ptr<Y> const & r);
    weak_ptr(weak_ptr const & r);
    template<class Y> weak_ptr(weak_ptr<Y> const & r);
    ~weak_ptr();
    weak_ptr & operator=(weak_ptr const & r);
    template<class Y> weak_ptr & operator=(weak_ptr<Y> const & r);
    template<class Y> weak_ptr & operator=(std::shared_ptr<Y> const & r);
    long use_count() const;
    bool expired() const;
    std::shared_ptr<T> lock() const;
};
```

Wykorzystanie weak_ptr

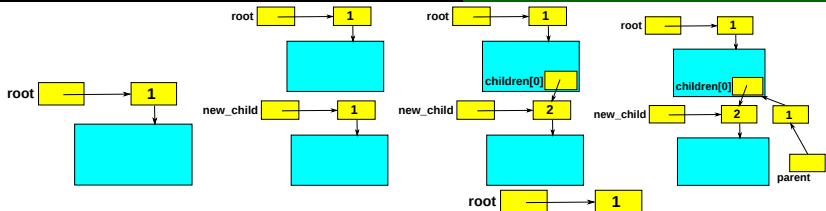
```
struct Foo {  
    std::weak_ptr<Foo> me_;  
};  
std::shared_ptr<Foo> pFoo = new Foo; //licznik równy 1  
pFoo->me_ = pFoo; //licznik równy 1, bo słaby wskaźnik  
//gdy pFoo zostanie zniszczone, to obiekt jest usuwany
```



sprytnie wskaźniki - błędne użycie

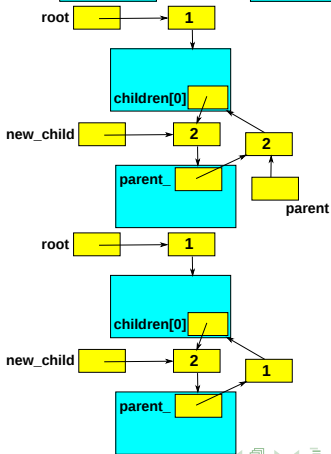
```
using PElement = std::shared_ptr<Element>;
class Element {
public:
    void setParent(PElement parent) {
        parent_ = parent;
    }
    void addChild(PElement new_child) {
        children_.push_back(new_child);
        new_child->setParent(PElement(this));
    }
    void removeChild() { children_.pop_back(); }
private:
    PElement parent_;
    std::vector<PElement> children_;
};

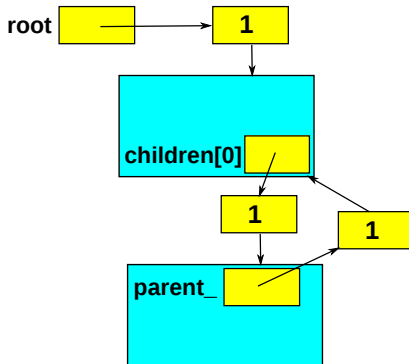
Element root(new Element);
root->addChild(PElement(new Element));
root->removeChild();//błąd! niszczy obiekt wskaz. przez root!
```



```
using PElement = std::shared_ptr<Element>;
class Element {
public:
    void setParent(PElement parent) {
        parent_ = parent;
    }
    void addChild(PElement new_child) {
        children_.push_back(new_child);
        new_child->setParent(PElement(this));
    }
    void removeChild() { children_.pop_back(); }
private:
    PElement parent_;
    std::vector<PElement> children_;
};

PElement root(new Element);
root->addChild(PElement(new Element));
root->removeChild();//błąd! niszczy obiekt
wskaz. przez root!
```





```
void addChild(PElement new_child) {  
    children_.push_back(new_child);  
    new_child->setParent(me_.lock() );  
}  
//dodatkowa składowa  
std::weak_ptr<Element> me_;
```

funkcja fabryczna `make_shared` i `allocate_shared`

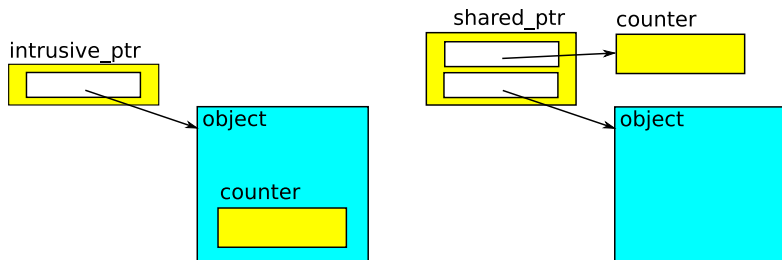
- ▶ eliminuje konieczność użycia operatora `new`
- ▶ jednocześnie alokuje obiekt i licznik (lepsze wykorzystanie pamięci)
- ▶ `make_shared` - woła globalny `::new`
- ▶ `allocate_shared` - używa dostarczonego alokatora

```
#include <boost/make_shared.hpp>
```

```
std::shared_ptr<Foo> pf = make_shared<Foo>(); //konstruktor domyślny  
std::shared_ptr<Foo> pf2 = make_shared<Foo>(arg1,...,argN);
```

boost::intrusive_ptr

Wykorzystuje licznik, który jest przechowywany w obiekcie.



funkcje pomocnicze:

- ▶ `intrusive_ptr_add_ref(T*)`; wołana przez konstruktory
- ▶ `intrusive_ptr_release(T*)` wołana przez destruktor

Tworzenie sprytnego wskaźnika na podstawie this

Wykorzystuje dziedziczenie po szablonie, dla którego typ jest parametrem (curiously recurring template pattern - omawiany szczegółowo na wykładzie o szablonych)

```
#include <boost/enable_shared_from_this.hpp>

class Node;

using PNode = boost::shared_ptr<Node>;

class Node: public boost::enable_shared_from_this<Node>{
    /* ... */
    void method(/* ... */) {
        PNode me = shared_from_this();
        /* ... */
    }
};
```

Sprytnie wskaźniki - podsumowanie

- ▶ brak przekazywania własności

```
//referencja  
void f(const Foo& f);
```

- ▶ ustalony właściciel - wskaźnik `std::unique_ptr`

```
std::unique_ptr<Foo> p = make_unique<Foo>(42);
```

- ▶ współwłaściciel - wskaźnik `std::shared_ptr`

```
std::shared_ptr<Foo> p = make_shared<Foo>(43);  
std::shared_ptr<Foo> q = p;
```

Uwaga: `shared_ptr` znacznie ($\approx 100\times$!) wolniejszy od `unique_ptr`

Sprytne wskaźniki - podsumowanie (2)

- ▶ `std::unique_ptr`
- ▶ `std::shared_ptr`,
- ▶ `std::weak_ptr`,
- ▶ `boost::scoped_ptr`, `boost::scoped_array`
- ▶ `boost::intrusive_ptr`

Rule of zero: zarządzanie zasobami przez sprytne wskaźniki

```
class Foo {  
    /* poprawny generowany konstruktor kopiujący (rule of five) */  
private:  
    shared_ptr<Goo> ptr_  
};
```

Dziękuję