# Zaawansowane CPP/Wykład 10: Inteligentne wskaźniki

### From Studia Informatyczne

< Zaawansowane CPP

## Spis treści



# Wstęp

Wskaźniki to jeden z bardziej pożytecznych elementów języka C/C++, ale na pewno najbardziej niebezpieczny. Zabawa z gołymi wskaźnikami przypomina żonglerkę odbezpieczonymi granatami. To nie jest już kwestia czy nastąpi wybuch, ale kiedy on nastąpi. Możliwości wywołania wybuchu i jego konsekwencje są wielorakie:

- Możemy usunąć wskaźnik nie zwalniając pamięci na którą on wskazuje. Jeśli żaden inny wskaźnik nie wskazuje na ten obszar to jest on bezpowrotnie stracony dla naszej aplikacji i mamy do czynienia z wyciekiem pamięci.
- Możemy zwolnić ten sam obszar pamięci kilkakrotnie. Powoduje to zwykle krach całej aplikacji.
- Podobnie jeśli spróbujemy zdereferencjonować wskaźniki `null`.
- No i oczywiście mamy całe spektrum możliwości sięgnięcia za pomocą wskaźnika tam gdzie nie powinniśmy, odczytując lub co gorsza zmieniając nie te zmienne co trzeba.

Jeśli więc nie czujemy się jak Rambo, albo nie przymierzając sam Chuck Norris, to powinniśmy poszukać jakichś zabezpieczeń. W C++ zabezpieczenia są dostarczane poprzez możliwość definicji własnych typów klas. Dzięki klasom możemy nie korzystać z dynamicznej alokacji pamięci bezpośrednio, ale za pośrednictwem klas, które dbają o alokacje w konstruktorach, dealokację w destruktorach, zwiększają i zmmniejszają pamięć na żądanie, itp. Przykładem takiego podejścia są np. kontenery STL, których jedną z zalet jest właśnie zarządzanie własną pamięcią. Jeśli jednak ciągle potrzebujemy wskaźników to możemy rozważyć opakowanie ich w klasy. Jest to możliwe dzięki możliwości jakie w C++ daje przeładowywanie operatorów, w szczególności możemy przeładowywać operatory dereferencjonowania `operator*()` i `operator->()`. W ten sposób możemy upodobnić zachowanie definiowanych przez nas typów do zachowania wskaźników. Takie typy nazywamy inteligentnymi wskaźnikami, ponieważ dostarczają nam dodatkowej funkcjonalności ponad zwykłe zachowanie wskaźnika.

Tak jak i u ludzi, rodzaje inteligencji wskaźników bywają różne i inteligentne wskaźniki występują w najróżniejszych wariacjach. Podział tych wariantów można przeprowadzić na wiele sposobów, ja skoncentruję sie na dwóch grupach:

- Zachowanie wskaźników podczas kopiowania, przypisywania i niszczenia. Nazwiemy to prawami własności.
- Zachowanie się operatorów `operator*()` i `operator->()`. Nazwiemy to kontrolą dostępu.

Poniżej krótko przedstawię przegląd głównych możliwości w każdej z powyższych grup.

# Prawa własności

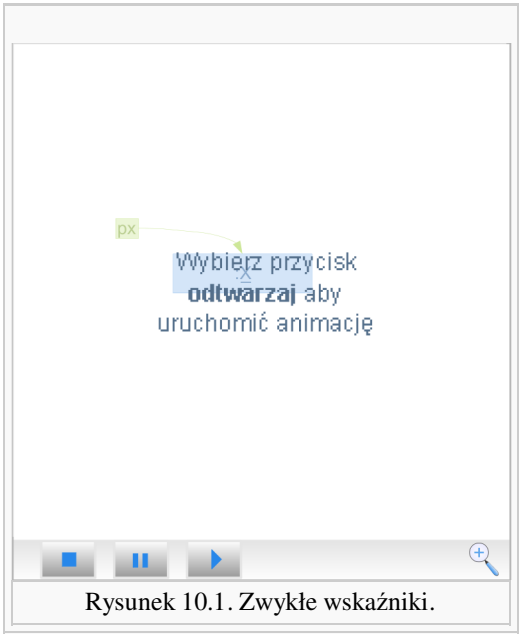
Głównym powodem używania inteligentnych wskaźników jest uzyskanie kontroli nad operacjami kopiowania, przypisywania i niszczenia wskaźnika. W tym kontekście mówimy często, że wskaźnik jest albo nie jest właścicielem obiektu na który wskazuje. Poniżej przedstawiam cztery typowe schematy wskaźników.

### Głupie wskaźniki

Zwykłe (nieinteligentne) wskaźniki, nie są właścicielami obiektów, na które wskazują. Kopiowanie czy przypisanie prowadzi do współdzielenia referencji (oba wskaźniki wskazują na ten sam obiekt) często niezamierzonej. Zniszczenie wskaźnika nie powoduje zniszczenia (dealokacji pamięci) obiektu, na który on wskazuje. Przedstawia to rysunek 10.1, na którym zilustrowano przebieg wykonania kodu:

```
void f() {
X *px( new X);
X  py(px);
X  pz(new X);
pz=py;
}

f();
```



Rysunek 10.1. Zwykłe wskaźniki.

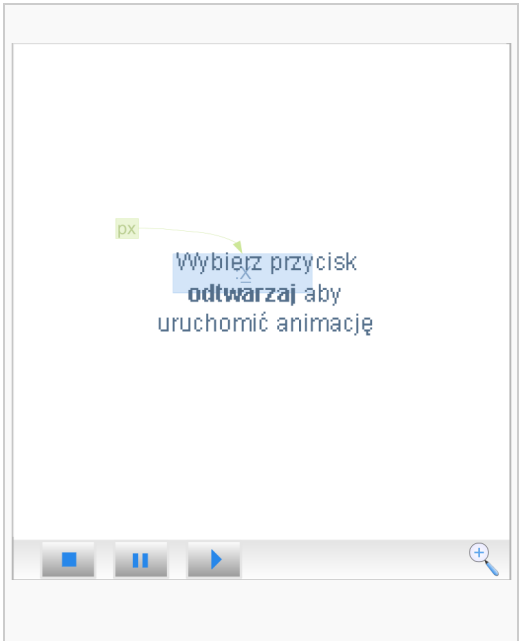
### Zliczanie referencji

Wskaźniki zliczające referencje są niejako właścicielami grupowymi obiektu, na który wskazują.

Kopiowanie i przypisanie powoduje współdzielnie referencji, ale kontrolowane, w tym sensie, że monitorowana jest liczba wskaźników do danego obiektu. Na zasadzie "ostatni gasi światło" zniszczenie wskaźnika powoduje zniszczenie obiektu wtedy gdy był to jedyny (ostatni) wskaźnik na ten obiekt. Liczenie referencji reprezentuje więc prostą wersję "odśmiecacza" (garbage-collector). Zachowanie się tego typu wskaźników prezentuje rysunek 10.2, w oparciu o analogiczny kod.

```
void f() {
ref_ptr<X>  px(new X);
ref_ptr<X>  py(px);
ref_ptr<X>  pz(new X);
pz=py;
}

f();
```



### Głęboka/fizyczna kopia

Rysunek 10.2. Wskaźniki zliczające referencje.

Takie wskaźniki są pojedynczymi właścicielami obiektów, na które wskazują i zachowują się jak wartości, a nie wskaźniki. Kopiowanie bądź przypisanie powoduje fizyczne kopiowanie obiektu wskazywanego. Zniszczenie wskaźnika powoduje zniszczenie wskazywanego obiektu. Od zwykłych wartości obiektów różnią się tym, że mają zachowanie polimorficzne i używane są tam gdzie polimorfizm jest nam potrzebny, a więc nie możemy użyć bezpośrednio samych obiektów. Zachowanie kodu

```
void f() {
clone_ptr<X>  px(new X);
clone_ptr<X>  py(px);
cloen_ptr<X>  pz(new X);
pz=py;
}

f();
```


Rysunek 10.3. Wskaźniki wykonujące kopie fizyczne.

ilustruje rysunek 10.3.

Zastosowanie wskaźników z głębokim kopiowaniem zilustruję na przykładzie podanego już wcześniej przykładu z kształtami geometrycznymi. W programie wykorzystującym takie kształty na pewno zachodzi konieczność kopiowania kształtów. Załóżmy, że wybraliśmy (myszką) jakiś kształt i wskaźnik do niego jest przechowywany w zmiennej `Shape *selected`. Załóżmy, że jest to obiekt typu `Circle`. Teraz chcemy uzyskać kopię tego kształtu. Przypisanie

```
Shape *copy=selected;
```

oczywiście nie zadziała, bo uzyskamy dwa wskaźniki na jeden obiekt. A my potrzebujemy drugiego obiektu. Bez konieczności polimorfizmu wystarczyłoby użyć konstruktora kopiującego:

```
Shape *copy=new Shape(*selected);
```

Niestety, w naszym przypadku ten kod się nawet nie skompiluje, bo klasa `Shape` jest klasą abstrakcyjną. Nawet gdyby nie była, to i tak zostałby utworzony obiekt typu `Shape`, a nie `Circle`. W celu zaimplementowania kopiowania polimorficznego możemy wyposażyć naszą klasę `Shape` w funkcję

```
virtual Shape *clone() const = 0
```

następnie zdefiniować ją w każdej podklasie:

```
class Circle :public Shape {
...
Circle *clone() {return new Circle(*this);};
}
```

i wtedy możemy skopiować (sklonować) nasz obiekt za pomocą

```
Shape *copy = selected->clone();
```

Możemy teraz tę technikę, nazywaną również wzorcem prototypu lub fabryką klonów (zob. E. Gamma, R. Helm, R. Johnson, J. Vlissides *"Wzorce projektowe. Elementy oprogramowania obiektowego wielokrotnego użytku"*), zastosować w implementacji inteligentnego wskaźnika `clone_ptr`

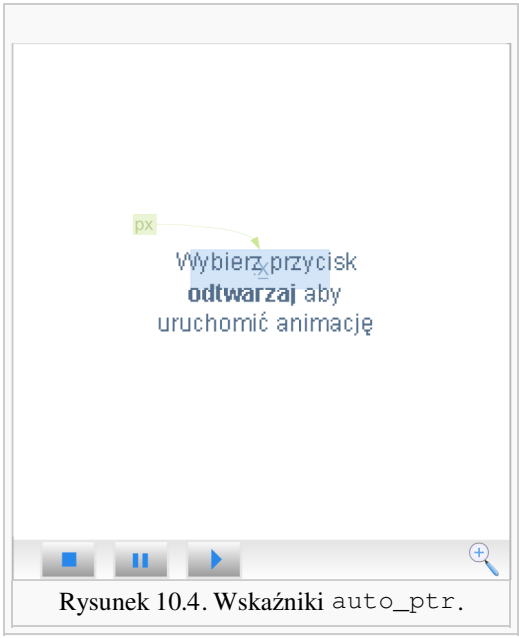
```
clone_ptr<Shape> selected;
...
clone_ptr<Shape> copy(selected);
```

**auto_ptr**

Wskaźniki `auto_ptr` (jedyne inteligentne wskaźniki dostępne w standardzie C++) są pojedynczymi, bardzo zaborczymi, właścicielami obiektu, na który wskazują. Tak zaborczymi, że nie dopuszczają możliwości współdzielenia obiektu ani jego kopiowania. Próba skopiowania albo przypisania prowadzi do przekazania własności: obiekt kopiowany(przypisywany) oddaje/ przekazuje prawo własności do posiadanego obiektu drugiemu obiektowi. Oznacza to, że obiekt kopiowany lub przypisywany jest zmieniany w trakcie tych operacji. Ilustruje to rysunek 10.4 na podstawie kodu

```
void f() {
auto_ptr<X>  px(new X);
auto_ptr<X>  py(px);
auto_ptr<X>  pz(new X);
pz=py;
}

f();
```



Rysunek 10.4. Wskaźniki `auto_ptr`.

To bardzo nieintuicyjne zachowanie: obiekty `auto_ptr` nie są modelami konceptu `Assignable`. Wskaźniki te zostały wprowadzone aby wspomagać bezpieczną alokację zasobów (głównie pamięci) według wzorca "alokacja zasobów jest inicjalizacją" (zob. B. Stroustrup *"Język C++"*). Rozważmy następujący przykład:

```
int  f() {
  BigX *p = new BigX;
... tu coś się dzieje
  delete  p;
  return wynik;
}
```

Jest to typowe zastosowanie dynamicznej alokacji pamięci. Problem polega na tym, że jeżeli pomiędzy przydziałem pamięci a jej zwolnieniem coś się stanie, to będziemy mieli wyciek pamięci. To coś to może być np. dodatkowe wyrażenie `return` lub rzucony wyjątek. W obu przypadkach zniszczone zostaną wszystkie statycznie zaalokowane obiekty, w tym i wskaźnik `p`. Ale ponieważ jest to zwykły wskaźnik jego zniszczenie nie spowoduje zwolnienia wskazywanej przez niego pamięci. Rozwiązaniem jest właśnie uczynienie go obiektem będącym właścicielem wskazywanej pamięci:

```
int  f() {
  auto_ptr<BigX> p(new BigX);

... tu coś się dzieje

  return wynik;
}
```

Teraz przy wyjściu z funkcji zostanie wywołany destruktor `p`, a on zwolni przydzieloną pamięć.

Wzkaźniki `auto_ptr` mogą być przekazywane i zwracane z funkcji. Jeśli przekażemy `auto_ptr` do funkcji przez wartość, to spowodowane tym kopiowanie spowoduje, że własność zostanie przekazana na argument funkcji i pamięć zostanie zwolniona kiedy funkcja zakończy swoje działanie.

```
template<typename T> void val(T p) {
};

auto_ptr<X> px(new X);
val(px);
px zawiera wskaźnik null. pamięć jest zwolniona
cout<<px.get()<<endl;
zwraca opakowany wskaźnik na X, powinien być zero
```

Jeśli przekażemy `auto_ptr` przez referencje to kopiowania nie będzie, przekazanie własności bedzie zależeć od tego czy wkaźnik zostanie skopiowany lub przypisany wewnątrz funkcji.

```
template<typename T> void ref_1(T &p) {
  T x = p;
};
template<typename T> void ref_2(T &p) {
};

auto_ptr<X> px(new X);
ref_2(px);
nic sie nie zmieniło
cout<<px.get()<<endl; wypisuje jakiś adres
ref_1(px)
px zawiera wskaźnik null. pamięć jest zwolniona
cout<<px.get()<<endl;
zwraca opakowany wskaźnik na X, powinien być zero
```

W przypadku przekazania `auto_ptr` jako referencji do stałej sprawa jest bardziej skomplikowana. Obecny standard stanowi, że wskaźnik `auto_ptr` przekazany jako referencja do stałej, nie przekazuje własności, tzn. operacje, które by do tego prowadziły nie skompilują się. Z tych samych powodów nie powinien skompilować się kod używający kontenerów STL zawierających wskaźniki `auto_ptr`.

```
template<typename T> void cref_1(const T &p) {
  T x = p;
};
template<typename T> void cref_2(const T &p) {
};

auto_ptr<X> px(new X);
cref_2(px);
OK, nic się nie stanie
cout<<px.get()<<endl; wypisuje jakiś adres
cref_1(px) nie skompiluje się

std::vector<auto_ptr<X> > v(10); nie skompiluje się
```

( Źródło: auto_ptr.cpp)

Różne implementacje różnie sobie z tym radzą i w praktyce wynik kompilowania powyższych fragmentów kodu może być różny na różnych platformach. Jest to dość techniczne zagadnienie, zainteresowane osoby odsyłam do D. Vandevoorde, N. Josuttis *"C++ Szablony, Vademecum profesjonalisty"* i N.M. Josuttis *"C++ Biblioteka standardowa, podręcznik programisty"*.

# Kontrola dostępu

Poza kontrolą rodzaju praw własności, inteligentny wskaźnik daje nam możliwość kontroli nad operacjami dostępu do wskazywanego obiektu poprzez operatory `operator->()` i `operator*()`. Wpływać na zachowanie tych operatorów możemy dwojako. Po pierwsze w oczywisty sposób możemy wykonać dodatkowy kod zanim zwrócimy z nich odpowiednią wartość:

```
T &operator*() {
zrób coś
return *_p;
}
```

Ten dodatkowy kod może np. sprawdzać czy wskaźnik `_p` nie jest zerowy, może zliczać wywołania, itp.

Po drugie możemy zmienić zwracany typ. Wbudowane operatory * i -> zwracają odpowiednio `T&` i `T*`. Oczywiście my możemy zwrócić cokolwiek, ale żeby to miało jakiś sens powinny to być obiekty zachowujące sie jak `T&` i `T*`. Takie obiekty które "zachowują się jak" coś, ale nie są tym (kwacze jak kaczka, ale to nie jest kaczka) nazywamy obiektami zastępczymi (proxy).

## Proxy

Dlaczego moglibyśmy chcieć używać obiektów zastępczych?

Typowe zastosowanie to implementacja operacji przypisania do obiektów, które tak naprawdę obiektami nie są. Weźmy jako przykład `ostream_iterator` dostarczany przez STL, który zezwala traktować plik wyjściowy jak kontener z iteratorem typu `OutputIterator`:

```
vector<int> V(10,7);
copy(V.begin(), V.end(), ostream_iterator<int>(cout, "\n"));
```

Przypatrzny się temu przykładowi bliżej. Jeśli zdefiniujemy

```
ostream_iterator<int>(cout, "\n") iout;
```

to w zasadzie jedyną dozwoloną operacją jest przypisanie i zwiększenie następujące po sobie:

```
(*iout) = 666; ++iout;
```

Ewidentnie nie istnieje żaden obiekt, do którego referencje moglibyśmy zwrócić. Możemy jednak zwrócić obiekt zastępczy, który będzie definiował operator przypisania:

```
class writing_proxy {
    std::ostream &_out;
  public:
    writing_proxy(std::ostream &out) :_out(out) {};

    void operator=(const T &val) {
      _out<<val;
    }
};
```

( Źródło: out.cpp)

Tę klasę zamkniemy wewnątrz klasy `ostream_iterator`

```
template<typename T> class ostream_iterator:
public std::iterator <std::output_iterator_tag, T >  {
```

```
  class writing_proxy {
  ...
  };

  std::string _sep;
  std::ostream &_out;
  writing_proxy  _proxy;
public:
  ostream_iterator(std::ostream &out,std::string sep):
    _out(out),_sep(sep),_proxy(_out){};
  void operator++()    {_out<<_sep;}
  void operator++(int) {_out<<_sep;}
  writing_proxy &operator*() {return _proxy;};
};
```

( Źródło: out.cpp)

Dziedziczenie z klasy `iterator` zapewni nam, że nasz `ostream_iterator` posiada wszystkie typy stowarzyszone wymagane przez iteratory STL. To z kolei pociąga za soba możliwość użycia `iterator_traits` (zob. wykład 5.5 (http://osilek.mimuw.edu.pl/index.php?title=Zaawansowane_CPP/Wyk%C5%82ad_5:_Klasy_cech#iterator_traits)). Bez tego nie moglibyśmy używać `ostream_iterator` w niektórych algorytmach STL.

Teraz możemy juz używać wyrażeń typu:

```
ostream_iterator<int> io(std::cout,"");
(*io)=44;
```

( Źródło: out.cpp)

Wywołanie `*io` zwraca `writing_proxy`. Następnie wywoływany jest

```
writing_proxy::operator=(44)}
```

który wykonuje operację

```
std::cout<<44;
```

Widać też, że operacja

```
i=(*io)
```

się nie powiedzie (nie skompiluje). W tym przypadku jest to pożądane zachowanie, bo taka operacja nie ma sensu. Jeśli byśmy jednak chcieli umożliwić działanie operacji przypisania w drugą stronę, możemy w obiekcie proxy zdefiniować operator konwersji na typ `T`.

```
operator T() {return T();}; uwaga bzdurny przykład !!!
```

Wtedy wykonanie

```
i=(*io)
```

przypisze zero do zmiennej `i`. W ten sposób obiekty proxy pozwalają nam rozróżniać użycie operatora `*` do odczytu i do zapisu.

Obiekty zastępcze stanowią zresztą często spotykany wzorzec projektowy (zob. E. Gamma, R. Helm, R. Johnson, J. Vlissides *"Wzorce projektowe. Elementy oprogramowania obiektowego wielokrotnego użytku"*). Poniżej przedstawię jeszcze jedną "sztuczkę" opisaną w A. Alexandrescu *"Nowoczesne Projektowanie w C++"*, służącą do automatycznego obudowywania funkcji wywoływanych za pośrednictwem inteligentnego wskaźnika wywoływaniami innych funkcji.

## Opakowywanie wywołań funkcji

Załóżmy, że mamy obiekt typu:

```
struct Widget {
  void pre() {cout<<"pre"<<endl;};
  void post() {cout<<"post"<<endl;};

  void f1() {cout<<"f1"<<endl;}
  void f2() {cout<<"f2"<<endl;}
}
```

( Źródło: pre_post.cpp)

Niech

```
Smart_prt<Widget> pw(new Widget);
```

bedzię inteligentnym wskaźnikiem do `Widget`. Chcemy aby każde wywołanie funkcji z `Widget` np.

```
pw->f1()
```

zostało poprzedzone przez wywołanie funkcji `pre()`, a po nim nastapiło wywołanie funkcji `post()`. Jedną z możliwości jest oczywiście zmiana kodu funkcji `f?`, tak aby wywoływały na początku `pre()` i `post()` na końcu. Można też dodać zestaw funkcji opakowywujących:

```
f1_wrapper() {pre();f1();post();}
```

Jest to jednak niepotrzebne duplikowanie kodu i możliwe do zastosowania tylko jeśli mamy możliwość zmiany kodu klasy `Widget`.

Można jednak zrobić inaczej. Zdefiniujemy pomocniczą klasę

```
template<typename T> struct Wrapper {
  T* _p;
  Wrapper(T* p):_p(p) {_p->pre();}
  ~Wrapper()          {_p->post();}

  T*  operator->() {return _p;}
};
```

( Źródło: pre_post.cpp)

W klasie inteligentnego wskaźnika przedefiniujemy `operator->()` tak, aby zwracał `Wrapper<T>(T *)` zamiast `T*`.

```
template<typename T> struct Smart_ptr {
  T *_p;
  Smart_ptr(T *p):_p(p) {};
  ~Smart_ptr(){delete _p;};
```

```
  Wrapper<T> operator->()   {return Wrapper<T>(_p);};
  T &operator*() {return *_p};
};
```

( Źródło: pre_post.cpp)

Jeśli teraz wywołamy

```
pw->f1();
```

to bedą się dziać następujace rzeczy:

- zostanie wywołany

```
pw.operator->();
```

operator ten zwraca obiekt `tmp` typu `Wrapper<Widget>`, ale najpierw musi go skonstruować, a więc

- zostanie wywołany konstruktor

```
tmp=Wrapper<Widget>(p);
```

który wywoła

```
p->pre();
```

- Jeśli `operator->()` zwróci obiekt, który posiada `operator->()` to jest on wywoływany rekurencyjnie, aż zostanie zwrócony typ wskaźnikowy. Tak więc zostanie wywołany `tmp.operator->()`, który zwroci `p`.
- Poprzez ten wskaźnik zostanie wywołana funkcja

```
p->f1();
```

- Zakończy się wywołanie `pw.operator->()`, a więc zostanie wywołany destruktor obiektu tymczasowego `tmp`, który wywoła

```
p->post();
```

Widać więc, że w końcu zostanie wykonana sekwencja wywołań:

```
p->pre();
p->f1();
p->post();
```

i tak bedzie dla dowolnej wywoływanej metody. Jeśli jednak wywołamy funkcję `f1()` za pomocą wyrażenia:

```
(*pw).f1();
```

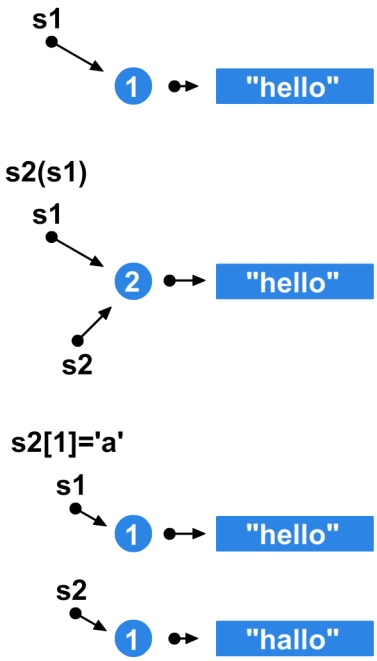
to ten mechanizm nie zadziała i nie ma możliwości, aby go w tej sytuacji zaimplementować. Może to być traktowane jako wada, bo nie jesteśmy w stanie zapewnić, że każde wywołanie funkcji zostanie opakowane, ale

z drugiej strony mamy do dyspozycji możliwość wyboru pomiędzy opakowanym i nieopakowanym wywołaniem funkcji.

### Współdzielenie reprezentacji

Opisując inteligentne wskaźniki nie można nie wspomnieć o technice implementacyjnej, która jest ściśle z nimi zwiazana, a mianowicie o współdzieleniu reprezentacji. Technika ta polega na oddelegowaniu całego (lub prawie całego) zachowania klasy do innego obiektu, nazywanego reprezentacją, a w obiekcie klasy przechowywanie tylko uchwytu do reprezentacji (zob rysunek 10.5).

```
class Wichajster {
public:
void do_something() {_rep->do_something();}
private:
  WichajsterRep _rep;
}
```




Rysunek 10.5.

Techniki tej używamy np. kiedy chcemy oszczędzić czas i miejsce potrzebne na kopiowanie obiektów. Kilka kopii obiektów klasy `Wichajster` może współdzielić jedną reprezentację korzystając np. ze zliczania referencji. Istotną różnicą w stosunku do inteligentnych wskaźników jest zachowanie w przypadku zmiany jednego z obiektów. W przypadku wskaźników, współdzielenie referencji jest planowaną cechą podejścia: kiedy zmieniamy obiekt poprzez jeden ze wskaźników wszystkie inne wskaźniki wskazują na zmieniony obiekt.

W przypadku współdzielenia reprezentacji chcemy cały czas rozróżniać obiekty, współdzielenie jest tylko technicznym środkiem optymalizacji. Wymaga to zastosowania techniki "copy on write", tzn. w momencie, w którym dokonujemy na obiekcie operacji mogącej go zmienić i jeśli posiada on współdzieloną reprezentację, to tworzymy nową fizyczna kopię tej reprezentacji i dopiero ją zmieniamy. Przedstawione to jest na rysunku 10.5. W przypadku metod łatwo stwierdzić, które zmieniaja obiekt, a które nie, problem jest tylko z metodami, które zwracaja referencje do wnętrza obiektu. Takie metody mogą służyć zarówno do zapisu, jak i do odczytu. Częściowym rozwiązaniem może być użycie obiektów proxy, tak jak to opisano w poprzednim podrozdziale. Szczegółowy opis tej techniki znajduje się w S. Meyers *"Język C++ bardziej efektywny"*.

# Iteratory

Iteratory to kolejny rodzaj inteligentnych wskaźników. Jeżli chodzi o prawa własności czy kontrolę dostępu to w większości zachowują się jak zwykłe wskaźniki. Wyjątkiem są specjalne iteratory, takie jak `ostream_iterator`, czy `back_inserter`, wspomniane powyżej. Ale zasadniczo inteligencja iteratorów umiejscowiona jest w operacjach arytmetycznych. Chodzi głównie o operator `operator++()` ponieważ wyposażone są w niego wszystkie iteratory kontenerów z STL. To właśnie jest zresztą podstawowa rola iteratora: przechodzenie do kolejnych elementów, semantyka wskaźnika to już wybór twórcow STL.

# Implementacje

Widać, że różnorodność inteligentnych wskaźników może przyprawić o zawrót głowy. A nie rozważyliśmy jeszcze wszystkich kwestii dotyczących ich zachowania. Wyczerpująca dyskusja na ten temat znajduje się w A. Alexandrescu *"Nowoczesne projektowanie"*. Tam też podana jest implemenatcja uniwersalnego szablonu klasy

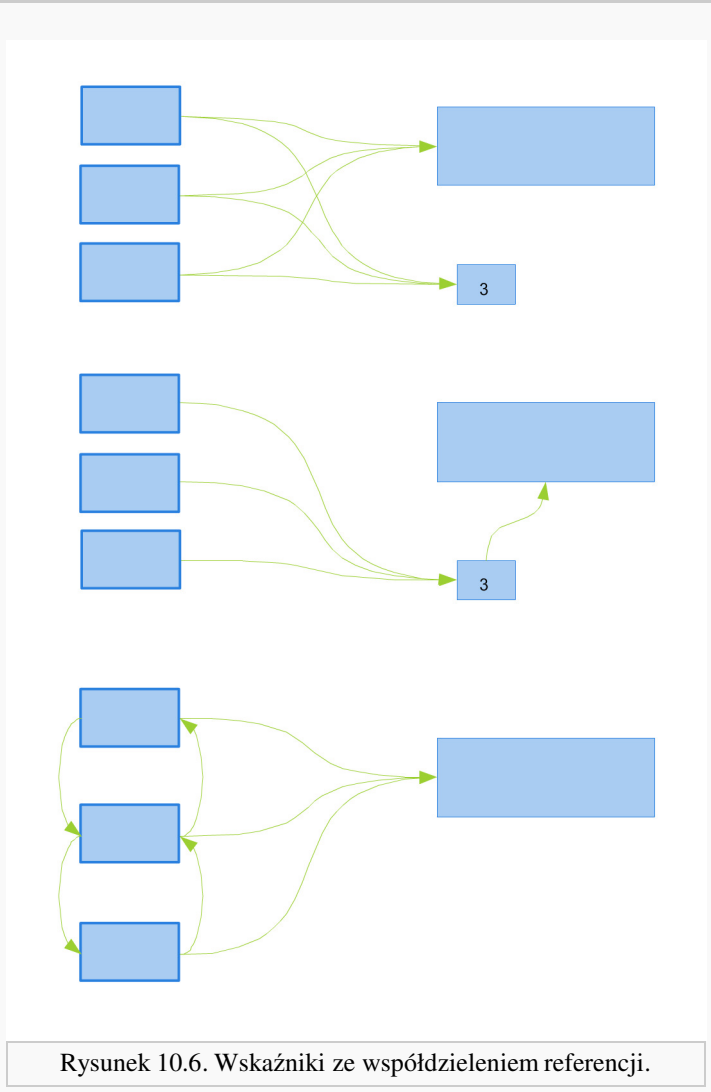
inteligentnego wskaźnika parametryzowanego kilkoma klasami wytycznymi. Alternatywą jest użycie szeregu klas (szablonów) implementujacych jeden typ wskaźnika każda. Zbiór takich klas można znaleźć w bibliotece `boost()`. Bardzo dobre opisy implementacji inteligentnych wskaźników znajdują się również w D. Vandevoorde, N. Josuttis: *"C++ Szablony, Vademecum profesjonalisty"* i S. Meyers *"Język C++ bardziej efektywny"*.

Tutaj dla przykładu zaprezentuję implementację wskaźnika zliczającego referencję paramtryzowanego jedną klasą wytyczną. Jest to podejście zbliżone do D. Vandevoorde, N. Josuttis: *"C++ Szablony, Vademecum profesjonalisty"*.

# Zliczanie referencji

Implementacje zliczania referencji różnią się przede wszystkim miejscem, w którym umieszczony zostanie licznik. Dwie główne możliwości to wewnątrz lub na zewnątrz obiektu, na który wskazujemy. Pierwsza możliwość jest ewidentnie możliwa tylko wtedy, jeśli mamy dostęp do kodu tej klasy. W każdej z tych dwu grup mamy dalsze możliwości, np.

1. Obiekt wskazywany udostępnia miejsce na licznik, zarządzaniem licznikiem zajmuje się wskaźnik
2. Obiekt wskazywany udostępnia nie tylko licznik, ale i interfejs do zarządzania nim.
3. Licznik jest osobnym obiektem. Każdy wskaźnik posiada wskaźnik na obiekt wskazywany i wskaźnik na licznik (zob. rysunek 10.6).
4. Licznik jest osobnym obiektem który zawiera również wskaźnik do obiektu wskazywanego. Każdy wskaźnik zawiera tylko wskaźnik do licznika (zob. rysunek 10.6).
5. Nie ma licznika, wskaźniki do tego samego obiektu połączone są w listę (zob. rysunek 10.6).



Rysunek 10.6. Wskaźniki ze współdzieleniem referencji.

Pokażę teraz przykladową implementację szablonu wskaźnika parametryzowanego jedną klasą wytyczną, określającą którąś z powyższych strategii, aczkolwiek przy jednej wytycznej jest to wysiłek, który pewnie sie nie opłaca, jako że kod wspólny jest dość mały. Ale implementacja ta może stanowić podstawę do rozszerzenia o kolejne wytyczne.

Najpierw musimy się zastanowić nad interfejsem lub raczej konceptem klasy wytycznej. W sumie najłatwiej to zrobić implemetując konkretną wytyczną. Zaczniemy od osobnego zewnętrznego licznika (zob. strategia 3). Klasa wytyczna musi zawierać wskaźnik do wspólnego licznika:

```
template<typename T> struct Extra_counter_impl {
...
private:
  size_t *_c;
};
```

i funkcje zwiekszające i zmniejszające licznik:

```
  public:
  bool remove_ref()     {--(*_c);return *_c==0;};
  void add_ref()        {++(*_c);};
  size_t count() {return *_c;};
};
```

Funkcja zmniejszająca licznik zwraca prawdę, jeśli usunięta została ostatnia referencja do wskazywanego obiektu. Potrzebna też będzie funkcja niszcząca licznik:

```
void cleanup() {
  delete _c;
  _c=0;
}
```

Potrzebne będą dwa konstruktory: defaultowy, który nic nie robi:

```
Extra_counter_impl():_c(0)                    {};
```

i konstruktor inicjalizujący licznik obiektu powstającego po raz pierwszy:

```
Extra_counter_impl(T* p):_c(new size_t) {*_c=0;};
```

który przydziela pamięć dla licznika. Argument `T* p` służy tylko do rozróżnienia tych konstruktorów.

Korzystając z tej klasy nietrudno jest napisać szablon inteligentnego wskaźnika. Obiekt licznika może być składową tego szablonu lub możemy dziedziczyć z klasy wytycznej (zob. wykład 7 (http://osilek.mimuw.edu.pl/index.php?title=Zaawansowane_CPP/Wyk%C5%82ad_7:_Klasy_wytycznych)). Niestety, okaże się, że bedziemy mieli problem próbując zaimplementować inne strategie, w szczególności strategię w której licznik i wskaźnik na obiekt wskazywany znajdują się w tym samym obiekcie (zob. strategia 4). Dlatego zmienimy trochę naszą implementację wytycznej i założymy, że obiekty tej klasy będą zawierać również wskaźnik na obiekt wskazywany

```
T *_p;
```

i dodamy funkcję:

```
T* pointee() {return _p;}
```

Musimy jeszcze poprawić funkcję czyszczącą:

```
void cleanup() {
  delete _c;
  delete _p;
  _p=0;
}
```

( Źródło: ref_ptr.h)

i jeden z konstruktorów:

```
Extra_counter_impl(T* p):_c(new size_t),_p(p) {*_c=0;};
```

Szablon wskaźnika korzystający z tak zdefiniowanej klasy wytycznej może wyglądać następująco:

```
template<typename T,
        typename counter_impl = Extra_counter_impl<T>  >
class Ref_ptr {
public:
  Ref_ptr() {};
  Ref_ptr(T *p):_c(p) {
    _c.add_ref();
  };

  ~Ref_ptr() {detach();}

  Ref_ptr(const Ref_ptr &p):_c(p._c) {
    _c.add_ref();
  }

  Ref_ptr &operator=(const Ref_ptr &rhs) {
    if(this!=&rhs) {
      detach();
      _c=rhs._c;
      _c.add_ref();
    }
    return *this;
  }

  T* operator->() {return _c.pointee();}
  T &operator*()  {return *(_c.pointee());}

  size_t count() {return _c.count();};
private:
  mutable counter_impl _c;
  void detach() {
    if (_c.remove_ref() ) _c.cleanup();
  };
};
```

( Źródło: ref_ptr.h)

# auto_ptr

Implementacja wskaźnika `auto_ptr` oparta jest o dwie funkcje. Jedna zwalnia przechowywany wskaźnik zwracając go na zewnątrz i oddając własność:

```
T* release()  {
  T *oldPointee = pointee;
  pointee = 0;
  return oldPointee;
}
```

( Źródło: auto_ptr.h)

`pointee` jest przechowywanym (zwykłym) wskaźnikiem.

```
private:
  T *pointee;
```

Druga funkcja zamienia przechowywany wskaźnik na inny, zwalniając wskazywaną przez niego pamięć:

```
void reset(T *p = 0)  {
  if (pointee != p) {
    delete pointee;
    pointee = p;
  }
}
```

( Źródło: auto_ptr.h)

Za pomocą tych funkcji można już łatwo zimplementować resztę szablonu, np.:

```
template<class T> class auto_ptr {
public:
  explicit auto_ptr(T *p = 0): pointee(p) {}

  template<class U>
  auto_ptr(auto_ptr <U> & rhs): pointee(rhs.release()) {}

  ~auto_ptr() { delete pointee; }

  template<class U>
  auto_ptr<T>& operator=(auto_ptr<U>& rhs)
  {
  if (this != &rhs) reset(rhs.release());
  return *this;
  }

  T& operator*() const { return *pointee; }
  T* operator->() const { return pointee; }
  T* get() const { return pointee; }
}
```

( Źródło: auto_ptr.h)

Konstruktor kopiujący i operator przypisania są szablonami, w ten sposób można kopiować również wskaźniki `auto_ptr` opakowujące typy, które mogą być na siebie rzutowane, np. można przypisać `auto_ptr<Derived>` do `auto_ptr<Base>`, jeśli `Derived` dziedziczy publicznie z `Base`. Konstruktor `auto_ptr(T *p = 0)` został zadeklarowany jako `explicit`, wobec czego nie spowoduje niejawnej konwresji z typu `T*` na `auto_ptr<T>`.

Różne impelentacje `auto_ptr` różnią się szczegółami dotyczącymi obsługi `const auto_ptr` i przekazywania `auto_ptr` przez stałą referencję. Powyższa implentacja wzięta z S. Meyers *"Język C++ bardziej efektywny"*, nie posiada pod tym względem żadnych zabezpieczeń. Szczegółowa dyskusja tego zagadnienia i bardziej zaawansowana implementacja znajduje się w N.M. Josuttis: *"C++ Biblioteka standardowa, podręcznik programisty"*. Temat ten jest też poruszony w D. Vandevoorde, N. Josuttis: *"C++ Szablony, Vademecum profesjonalisty"*. Warto też zaglądnąć do implementacji `auto_ptr` dostarczonej z kompilatorem `g++`.