Listy różnych obiektów

Bogdan Kreczmer

bogdan.kreczmer@pwr.edu.pl

Katedra Cybernetyki i Robotyki
Politechnika Wrocławska

Kurs: Programowanie obiektowe

Niniejsza prezentacja została wykonana przy użyciu systemu składu LaTeX *oraz stylu* `beamer`, *którego autorem jest Till Tantau.*

Strona domowa projektu `Beamer`:
`http://latex-beamer.sourceforge.net`

Plan prezentacji

Plan prezentacji

Lista obiektów (std::list<>)

```
class ObiektGraf {
  public:
   ObiektGraf() { cout << "+++ Konstruktor" << endl; }
   ~ObiektGraf() { cout << "--- Destruktor" << endl; }
};

int main()
{
  ObiektGraf                ObGraf;
  std::list<ObiektGraf>     Lst;

  cout << "== Przed ==========" << endl;
  Lst.push_back(ObGraf);
  Lst.push_back(ObGraf);
  Lst.push_back(ObGraf);
  cout << "== Po ==========" << endl;
}
```

Lista obiektów (std::list<>)

```
class ObiektGraf {
  public:
   ObiektGraf() { cout << "+++ Konstruktor" << endl; }
   ~ObiektGraf() { cout << "--- Destruktor" << endl; }
};

int main()
{
  ObiektGraf                ObGraf;
  std::list<ObiektGraf>     Lst;

  cout << "== Przed ============" << endl;
  Lst.push_back(ObGraf);
  Lst.push_back(ObGraf);
  Lst.push_back(ObGraf);
  cout << "== Po ===============" << endl;
}
```

Wynik działania:

```
+++ Konstruktor
== Przed ============
== Po ===============
--- Destruktor
--- Destruktor
--- Destruktor
--- Destruktor
```

Lista obiektów (std::list<>)

```
class ObiektGraf {
  public:
   ObiektGraf() { cout << "+++ Konstruktor" << endl; }
   ~ObiektGraf() { cout << "--- Destruktor" << endl; }
};

int main()
{
  ObiektGraf                ObGraf;
  std::list<ObiektGraf>     Lst;

  cout << "== Przed ============" << endl;
  Lst.push_back(ObGraf);
  Lst.push_back(ObGraf);
  Lst.push_back(ObGraf);
  cout << "== Po ===============" << endl;
}
```

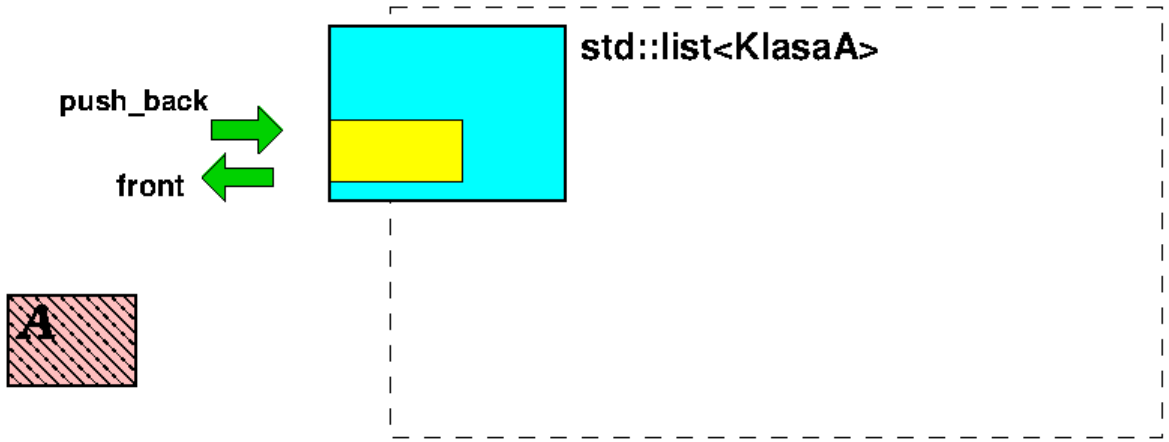
:-O

Gdzie są brakujące obiekty???!!!

Wynik działania:

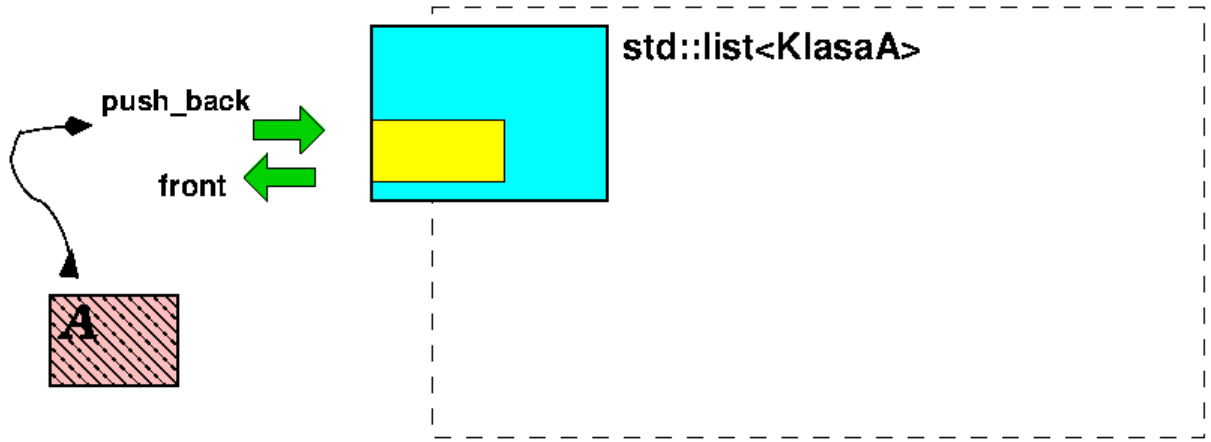
```
+++ Konstruktor
== Przed ============
== Po ===============
--- Destruktor
--- Destruktor
--- Destruktor
--- Destruktor
```

Lista obiektów (std::list<>) – jak to działa



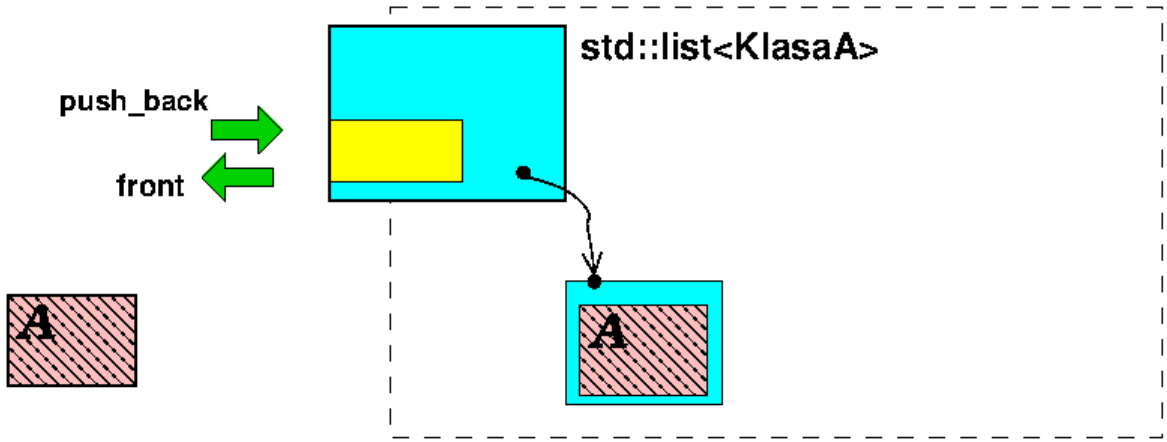

```
int main()
{
  ObiektGraf                ObGraf;
  std::list<ObiektGraf>     Lst;

  cout << "== Przed ==========" << endl;
  Lst.push_back(ObGraf);
  Lst.push_back(ObGraf);
  Lst.push_back(ObGraf);
  cout << "== Po ==========" << endl;
}
```

Lista obiektów (std::list<>) – jak to działa



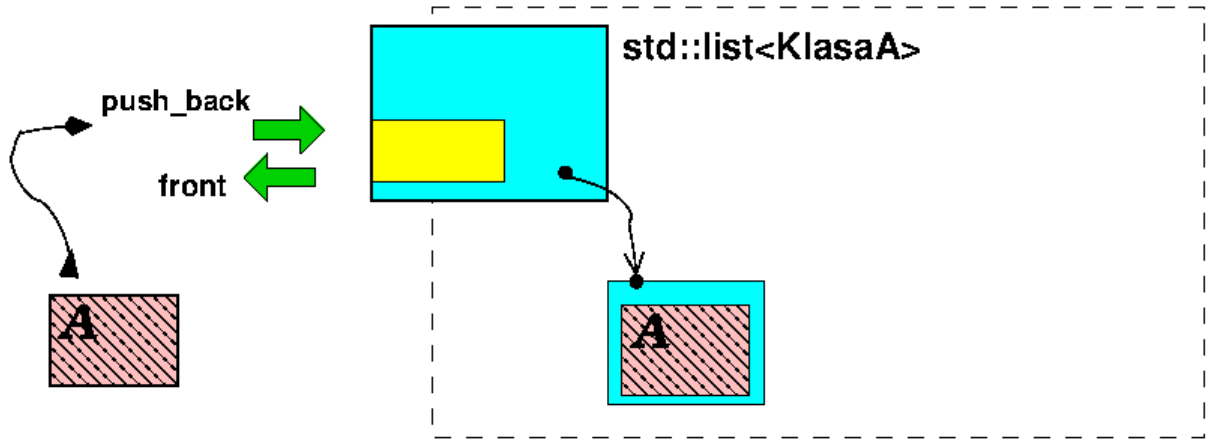

```
int main()
{
  ObiektGraf                ObGraf;
  std::list<ObiektGraf>     Lst;

  cout << "== Przed ==========" << endl;
  Lst.push_back(ObGraf);    <=
  Lst.push_back(ObGraf);
  Lst.push_back(ObGraf);
  cout << "== Po ==========" << endl;
}
```

Lista obiektów (std::list<>) – jak to działa



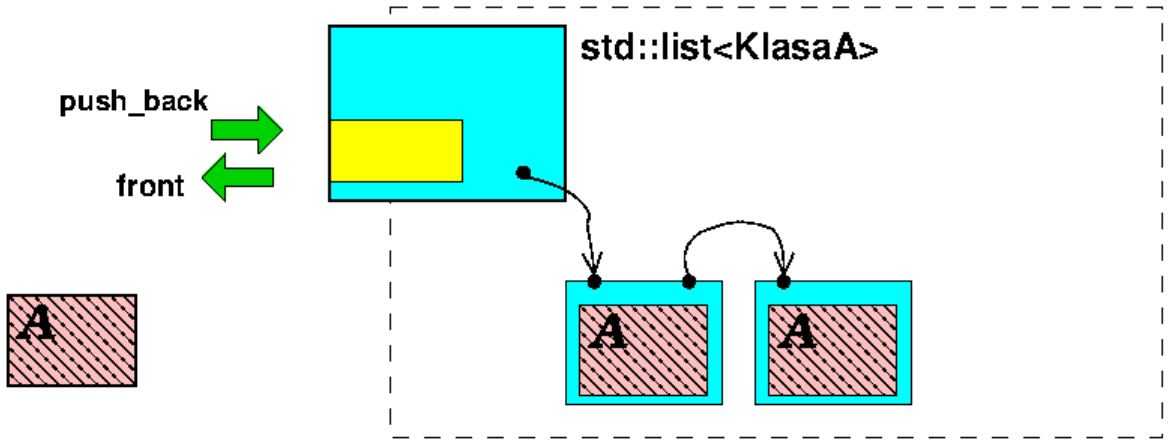

```
int main()
{
  ObiektGraf                ObGraf;
  std::list<ObiektGraf>     Lst;

  cout << "== Przed ==========" << endl;
  Lst.push_back(ObGraf);    <=
  Lst.push_back(ObGraf);
  Lst.push_back(ObGraf);
  cout << "== Po ==========" << endl;
}
```

Lista obiektów (std::list<>) – jak to działa



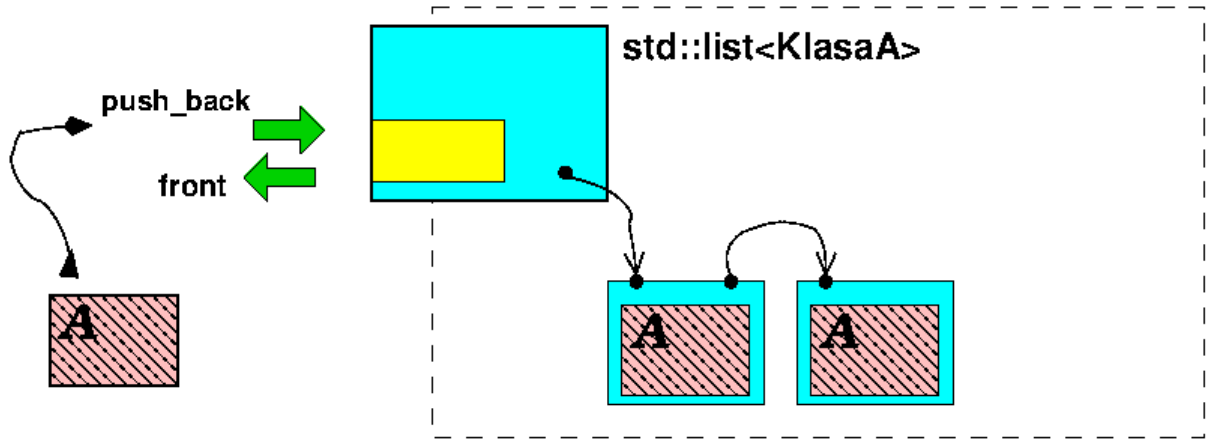

```
int main()
{
  ObiektGraf                ObGraf;
  std::list<ObiektGraf>     Lst;

  cout << "== Przed ==========" << endl;
  Lst.push_back(ObGraf);
  Lst.push_back(ObGraf);    <=
  Lst.push_back(ObGraf);
  cout << "== Po ==========" << endl;
}
```

Lista obiektów (std::list<>) – jak to działa



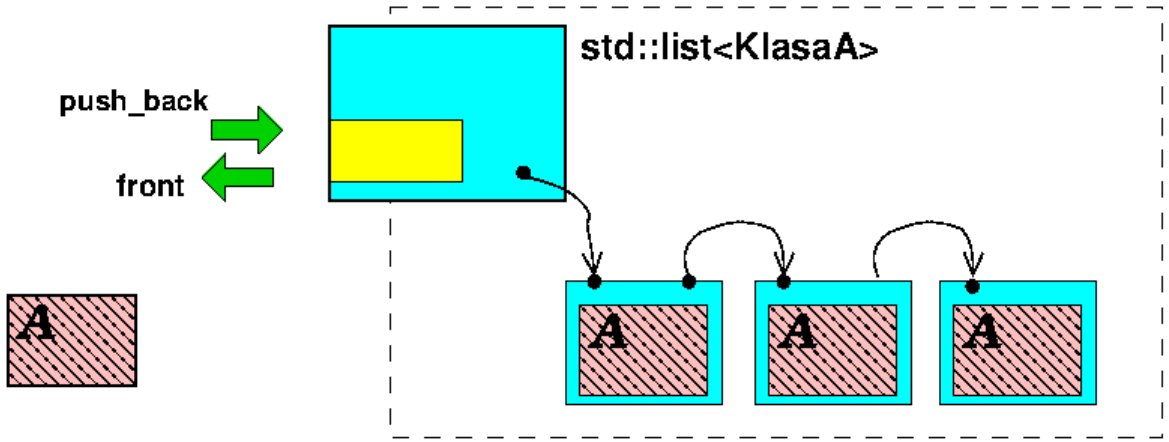

```
int main()
{
  ObiektGraf                ObGraf;
  std::list<ObiektGraf>     Lst;

  cout << "== Przed ==========" << endl;
  Lst.push_back(ObGraf);
  Lst.push_back(ObGraf);    <=
  Lst.push_back(ObGraf);
  cout << "== Po ==========" << endl;
}
```

Lista obiektów (std::list<>) – jak to działa



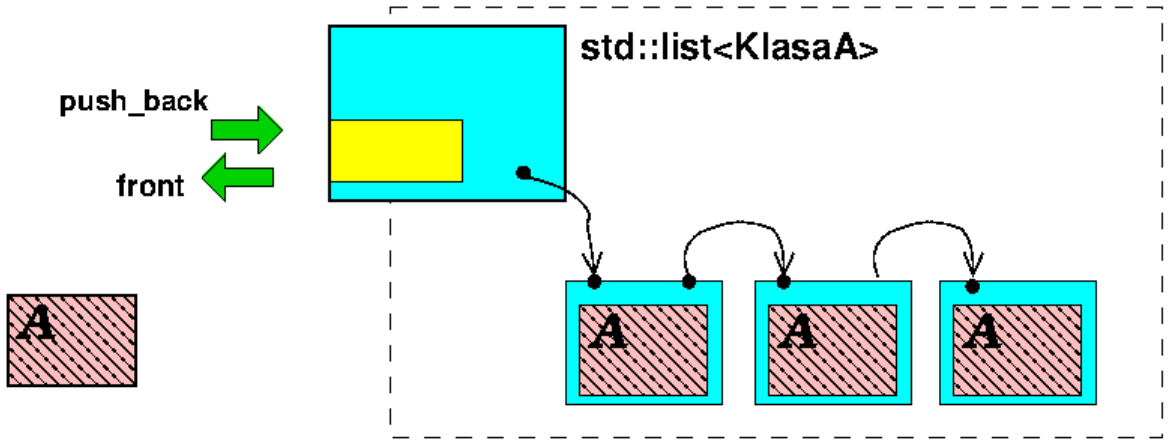

```
int main()
{
  ObiektGraf                ObGraf;
  std::list<ObiektGraf>     Lst;

  cout << "== Przed ==========" << endl;
  Lst.push_back(ObGraf);
  Lst.push_back(ObGraf);
  Lst.push_back(ObGraf);    <=
  cout << "== Po ==========" << endl;
}
```

Lista obiektów (std::list<>) – jak to działa

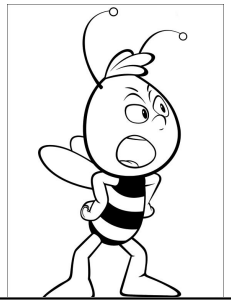



```
int main()
{
  ObiektGraf                ObGraf;
  std::list<ObiektGraf>     Lst;

  cout << "== Przed ==========" << endl;
  Lst.push_back(ObGraf);
  Lst.push_back(ObGraf);
  Lst.push_back(ObGraf);    <=
  cout << "== Po ==========" << endl;
}
```

Lista obiektów (std::list<>) – jak to działa




```
int main()
{
  ObiektGraf                ObGraf;
  std::list<ObiektGraf>     Lst;

  cout << "== Przed ============" << endl;
  Lst.push_back(ObGraf);
  Lst.push_back(ObGraf);
  Lst.push_back(ObGraf);
  cout << "== Po ==============" << endl;
}
```



Wynik działania:

```
+++ Konstruktor
== Przed ============
== Po ==============
--- Destruktor
--- Destruktor
--- Destruktor
--- Destruktor
```

Lista obiektów – konstruktor kopiujący

```
class ObiektGraf {
  public:
   ObiektGraf()  { cout << "+++ Konstruktor" << endl; }
   ObiektGraf(const ObiektGraf& )
                 { cout << "+++ Kopiuje" << endl; }
   ~ObiektGraf() { cout << "--- Destruktor" << endl; }
};

int main()
{
  ObiektGraf                ObGraf;
  std::list<ObiektGraf>     Lst;

  cout << "== Przed ==========" << endl;
  Lst.push_back(ObGraf);
  Lst.push_back(ObGraf);
  Lst.push_back(ObGraf);
  cout << "== Po ==========" << endl;
}
```

Lista obiektów – konstruktor kopiujący

```
class ObiektGraf {
  public:
   ObiektGraf()  { cout << "+++ Konstruktor" << endl; }
   ObiektGraf(const ObiektGraf& Ob)
                 { (*this) = Ob; cout << "+++ Kopiuje" << endl; }
   ~ObiektGraf() { cout << "--- Destruktor" << endl; }
};

int main()
{
  ObiektGraf                ObGraf;
  std::list<ObiektGraf>     Lst;

  cout << "== Przed ==========" << endl;
  Lst.push_back(ObGraf);
  Lst.push_back(ObGraf);
  Lst.push_back(ObGraf);
  cout << "== Po ==========" << endl;
}
```

W przypadku, gdy w danej klasie są jakieś pola, definiując własny konstruktor kopiujący tylko po to, aby wyświetlić ślad jego działania, nie można zapominać, aby wykonać operację kopiowania.

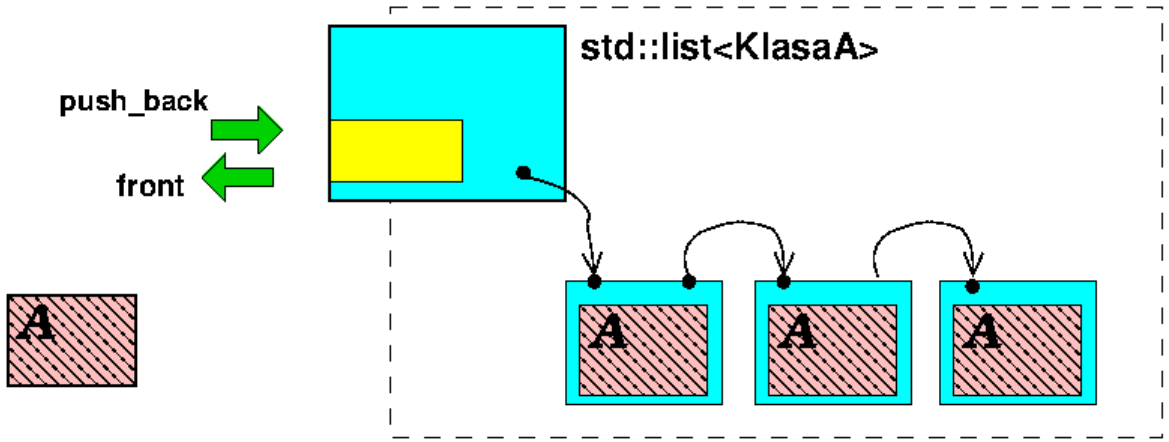
Lista obiektów – konstruktor kopiujący

```
class ObiektGraf {
  public:
   ObiektGraf()  { cout << "+++ Konstruktor" << endl; }
   ObiektGraf(const ObiektGraf& Ob)
                 { (*this) = Ob; cout << "+++ Kopiuje" << endl; }
   ~ObiektGraf() { cout << "--- Destruktor" << endl; }
};

int main()
{
  ObiektGraf                ObGraf;
  std::list<ObiektGraf>     Lst;

  cout << "== Przed ============" << endl;
  Lst.push_back(ObGraf);
  Lst.push_back(ObGraf);
  Lst.push_back(ObGraf);
  cout << "== Po ===============" << endl;
}
```

Wynik działania:

```
+++ Konstruktor
== Przed ============
+++ Kopiuje
+++ Kopiuje
+++ Kopiuje
== Po ===============
--- Destruktor
--- Destruktor
--- Destruktor
--- Destruktor
```

Lista obiektów – wynik końcowy




```
int main()
{
  ObiektGraf                ObGraf;
  std::list<ObiektGraf>     Lst;

  cout << "== Przed ============" << endl;
  Lst.push_back(ObGraf);
  Lst.push_back(ObGraf);
  Lst.push_back(ObGraf);
  cout << "== Po ==============" << endl;
}
```

Wynik działania:

```
+++ Konstruktor
== Przed ============
+++ Kopiuje
+++ Kopiuje
+++ Kopiuje
== Po ==============
--- Destruktor
--- Destruktor
--- Destruktor
--- Destruktor
```

Plan prezentacji

Przykładowa hierarchia dziedziczenia

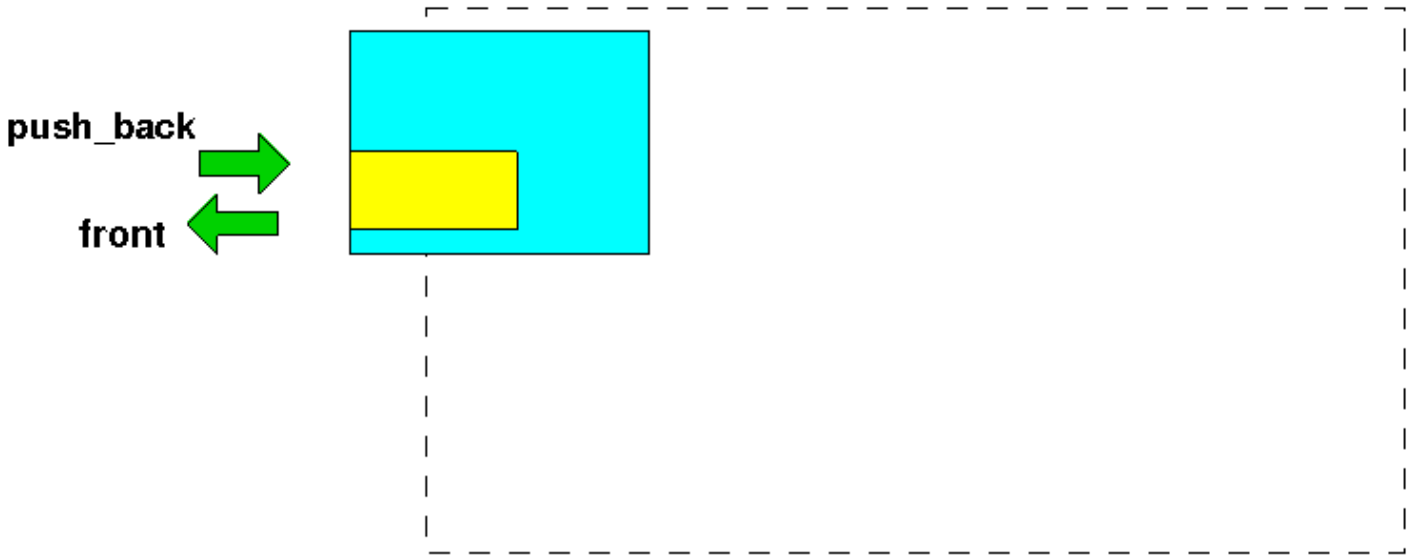
```
class ObiektGraficzny {
  ...
};

class Robot: public: ObiektGraficzny {
  ...
};

class Sciezka: public: ObiektGraficzny {
  ...
};

class Przeszkoda: public: ObiektGraficzny {
  ...
};
```

Przykładowa hierarchia dziedziczenia

```
int main()
{
  std::list<ObiektGraficzny>  Lst;
  Robot                       Rob;
  Sciezka                     Sci;
  Przeszkoda                  Prz;

  cout << "== Przed ==========" << endl;
  Lst.push_back(Rob);
  Lst.push_back(Sci);
  Lst.push_back(Prz);
  cout << "== Po ==========" << endl;
}
```

Dzięki dziedziczeniu zachodzić będzie niejawne rzutowanie na klasę bazową. Tak więc niniejszy kod zostanie zaakceptowany przez kompilator. Jednak czy doprowadzi on do powstania listy różnych obiektów?

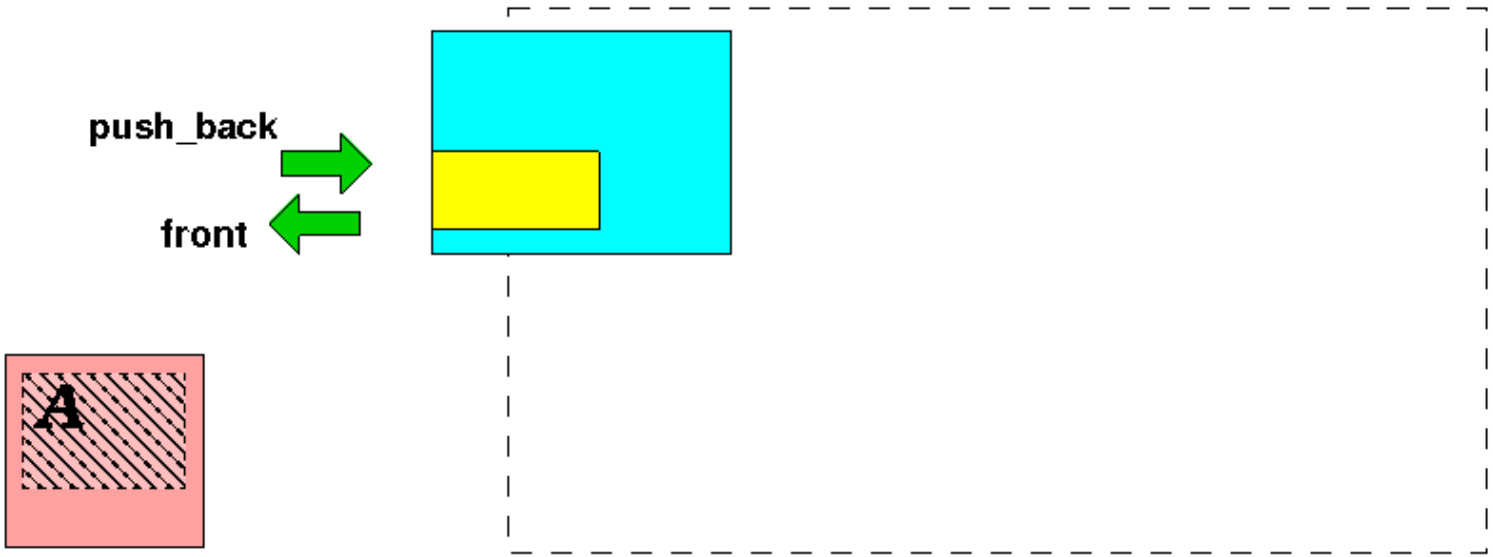
Lista obiektów `std::list<ObiektGraficzny>`



W liście można *próbować* umieścić obiekty klas pochodnych takich jak np. `Robot`, `Sciezka`, `Przeszkoda`.

Jednak do listy przekopiowane zostaną tylko podobiekty klasy bazowej.

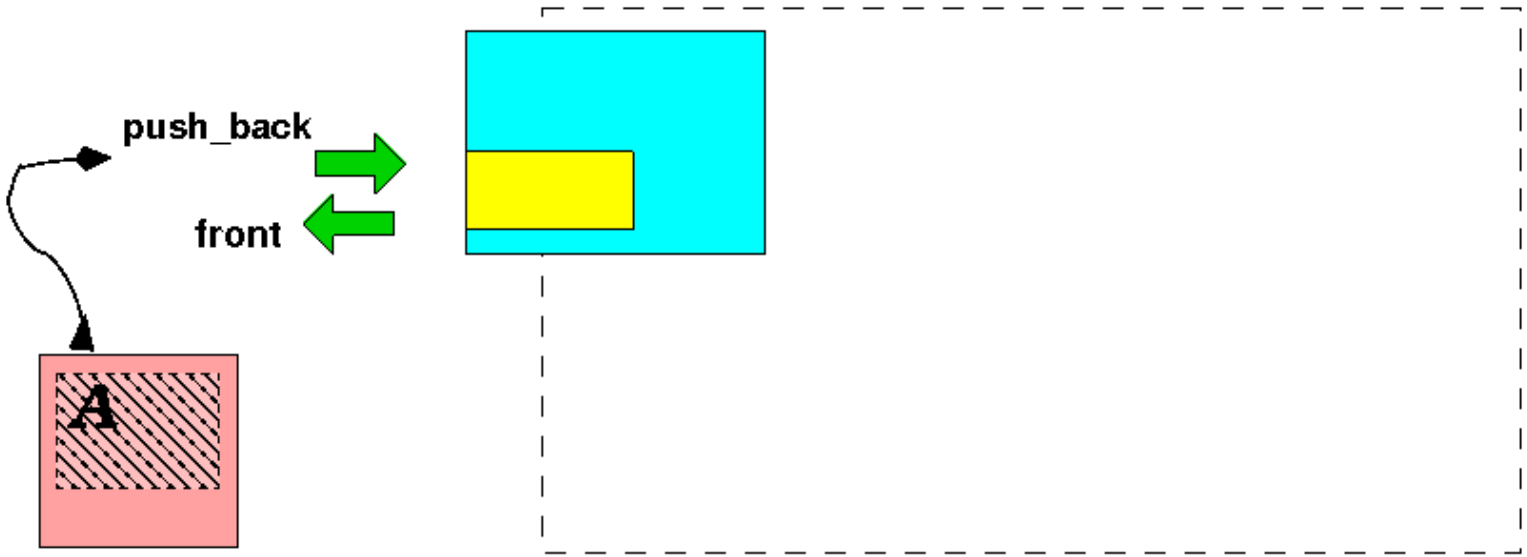
Lista obiektów `std::list<ObiektGraficzny>`



W liście można *próbować* umieścić obiekty klas pochodnych takich jak np. `Robot`, `Sciezka`, `Przeszkoda`.

Jednak do listy przekopiowane zostaną tylko podobiekty klasy bazowej.

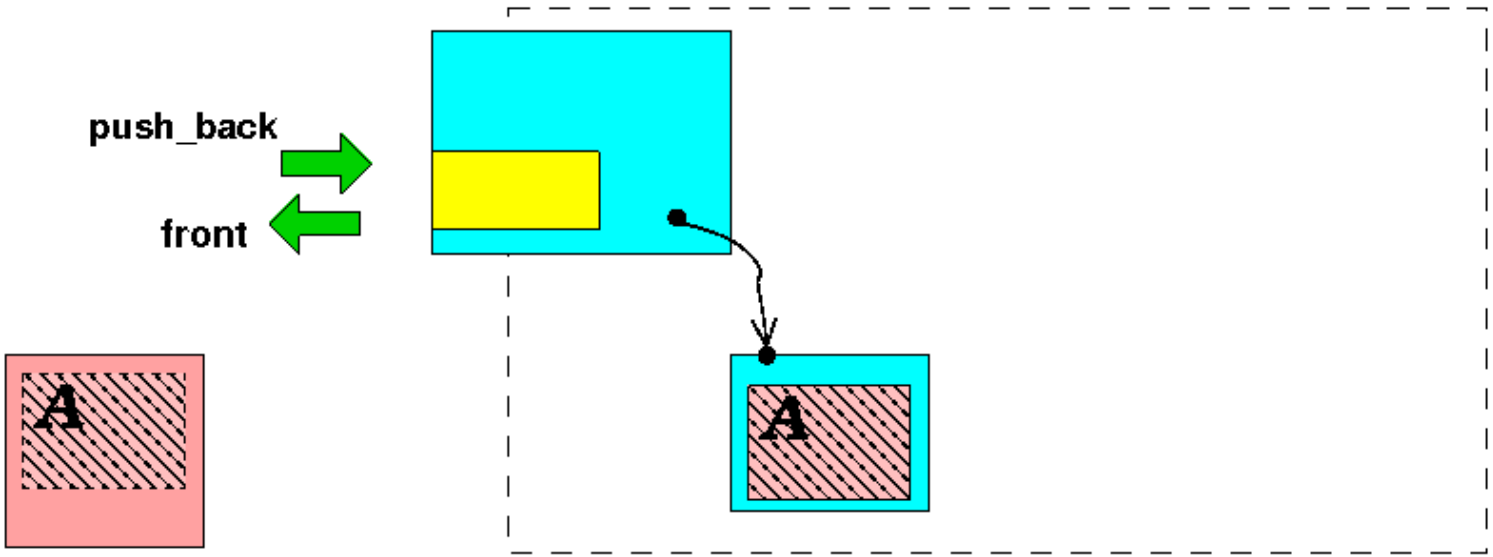
Lista obiektów `std::list<ObiektGraficzny>`



W liście można *próbować* umieścić obiekty klas pochodnych takich jak np. `Robot`, `Sciezka`, `Przeszkoda`.

Jednak do listy przekopiowane zostaną tylko podobiekty klasy bazowej.

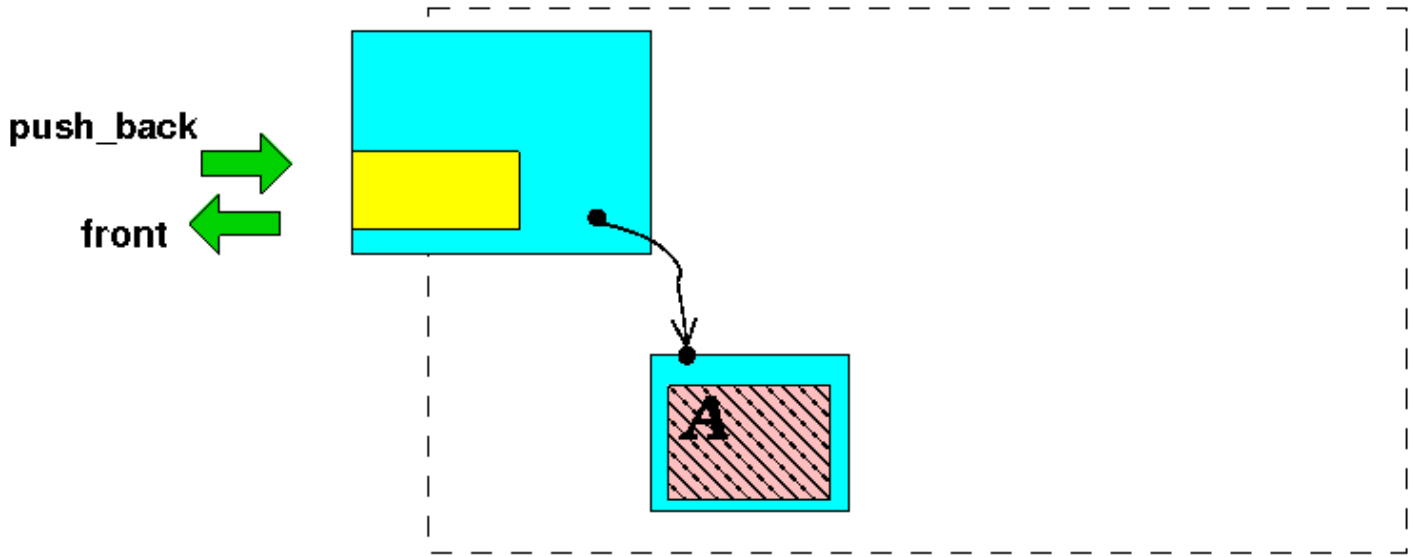
Lista obiektów `std::list<ObiektGraficzny>`

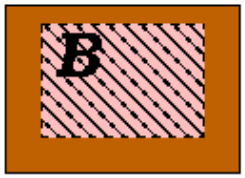


W liście można *próbować* umieścić obiekty klas pochodnych takich jak np. `Robot`, `Sciezka`, `Przeszkoda`.

Jednak do listy przekopiowane zostaną tylko podobiekty klasy bazowej.

Lista obiektów `std::list<ObiektGraficzny>`

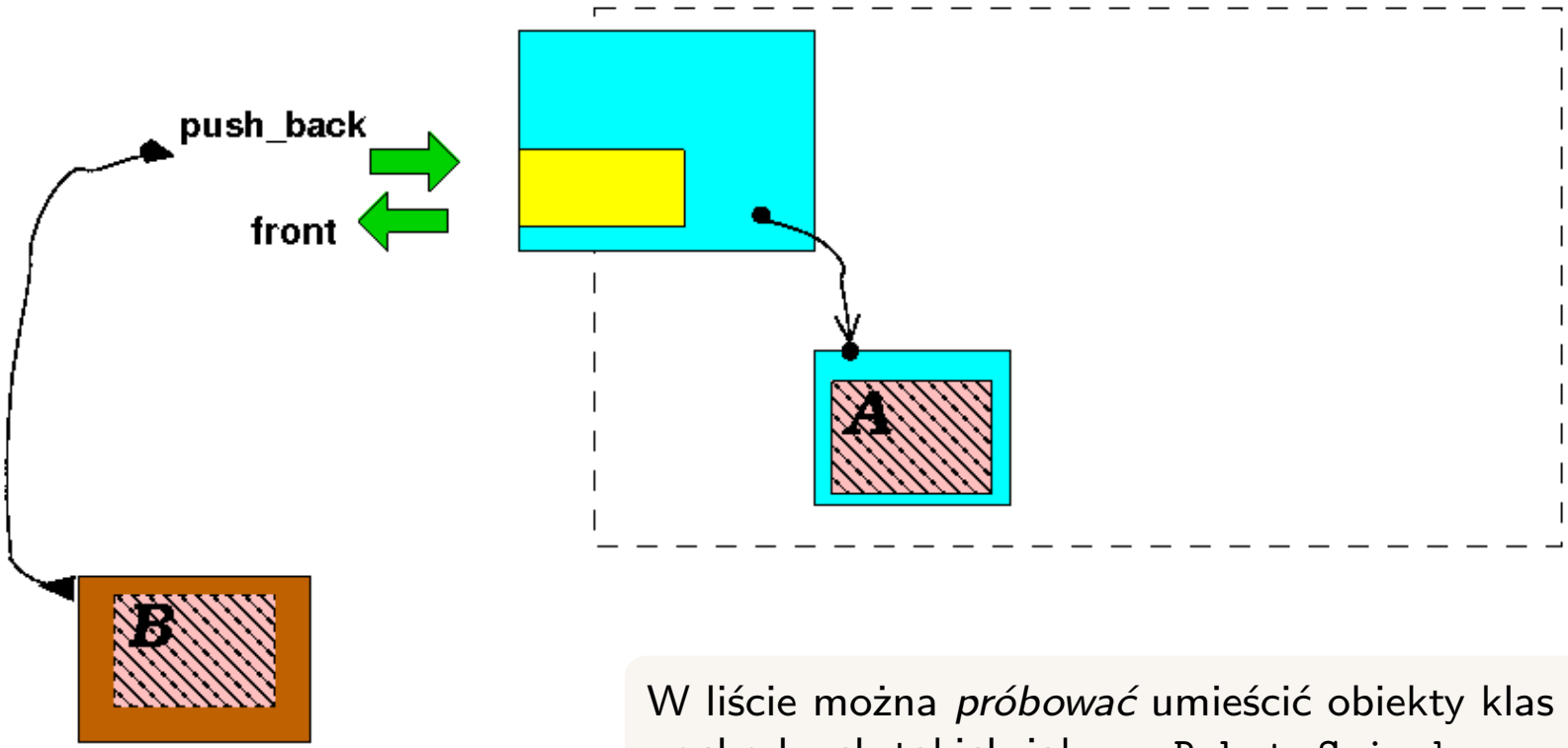




W liście można *próbować* umieścić obiekty klas pochodnych takich jak np. `Robot`, `Sciezka`, `Przeszkoda`.

Jednak do listy przekopiowane zostaną tylko podobiekty klasy bazowej.

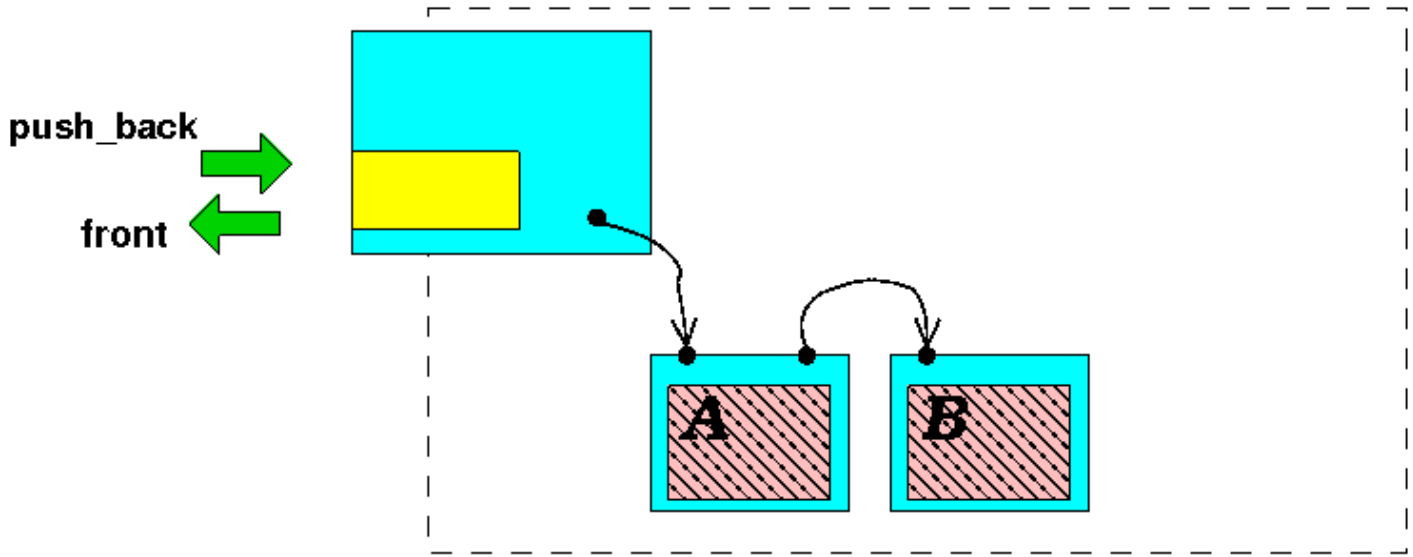
Lista obiektów `std::list<ObiektGraficzny>`

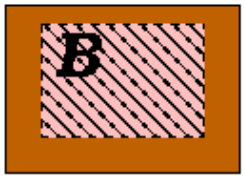


W liście można *próbować* umieścić obiekty klas pochodnych takich jak np. `Robot`, `Sciezka`, `Przeszkoda`.

Jednak do listy przekopiowane zostaną tylko podobiekty klasy bazowej.

Lista obiektów `std::list<ObiektGraficzny>`

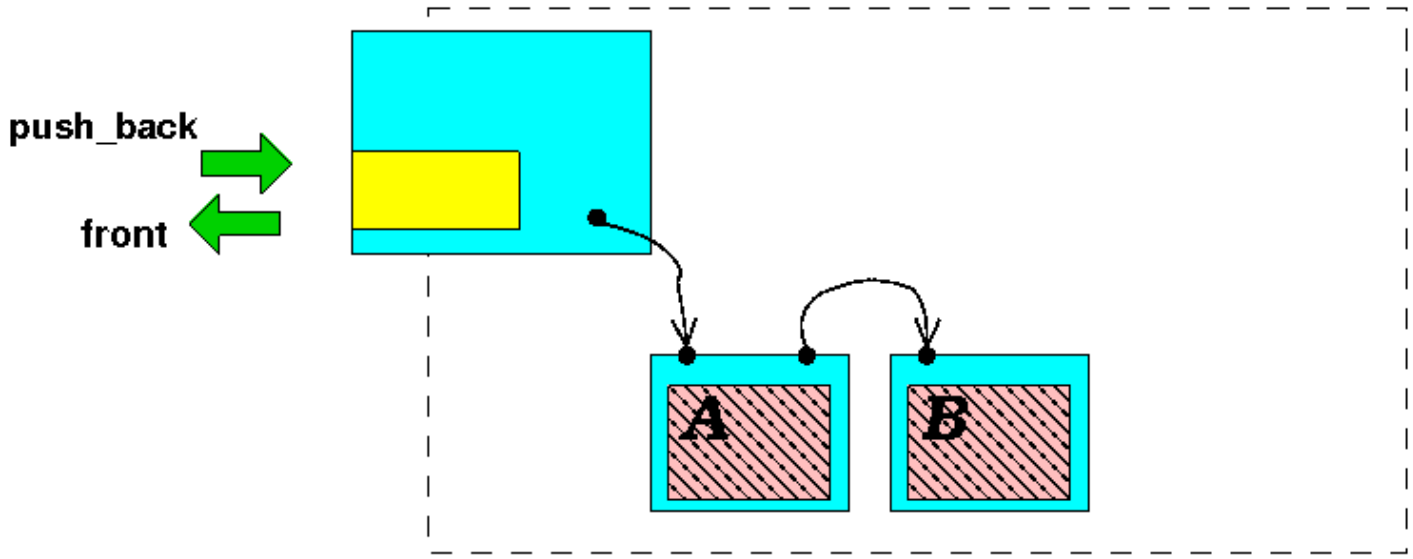


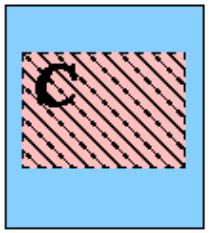


W liście można *próbować* umieścić obiekty klas pochodnych takich jak np. `Robot`, `Sciezka`, `Przeszkoda`.

Jednak do listy przekopiowane zostaną tylko podobiekty klasy bazowej.

Lista obiektów `std::list<ObiektGraficzny>`

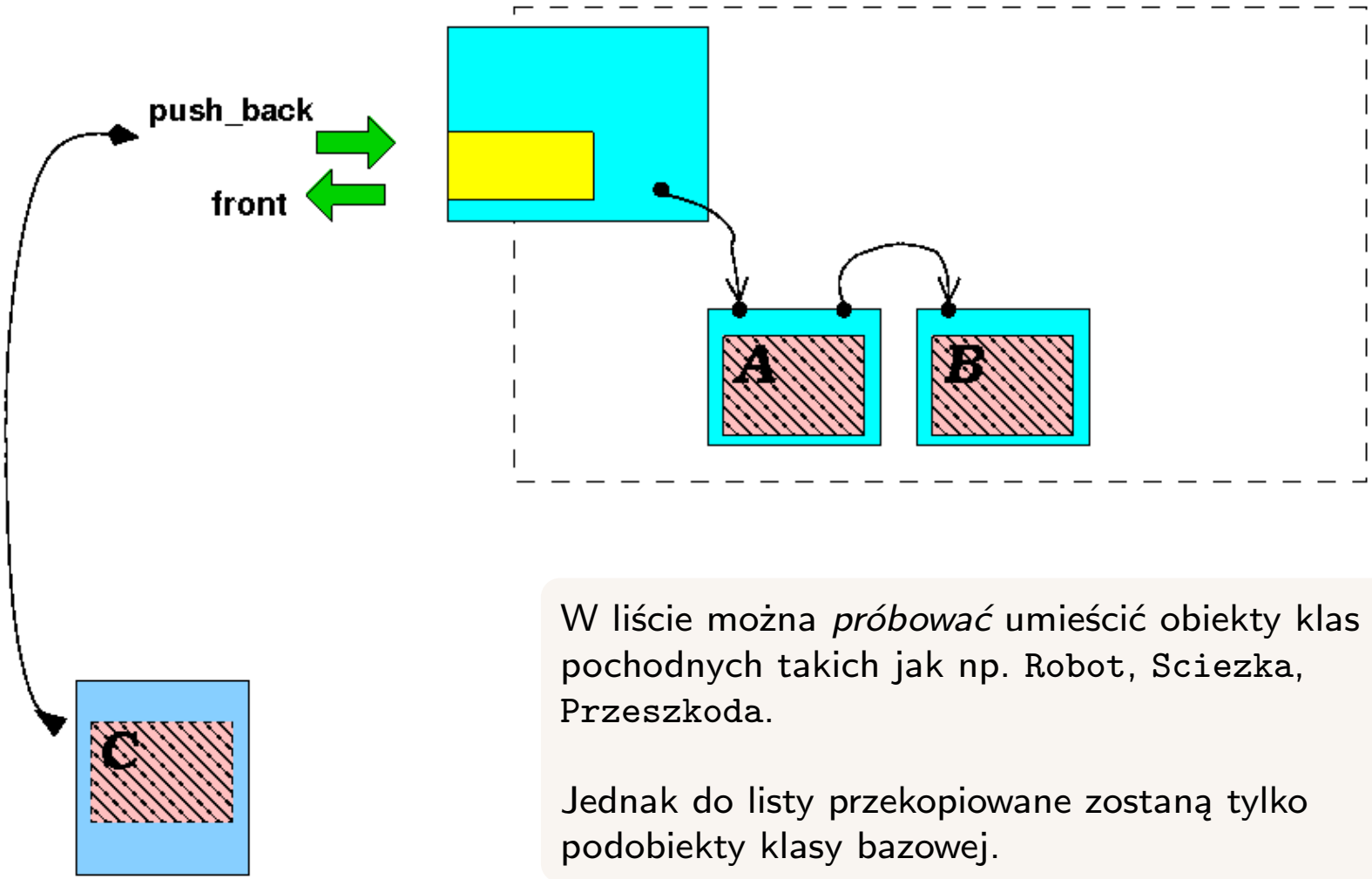




W liście można *próbować* umieścić obiekty klas pochodnych takich jak np. `Robot`, `Sciezka`, `Przeszkoda`.

Jednak do listy przekopiowane zostaną tylko podobiekty klasy bazowej.

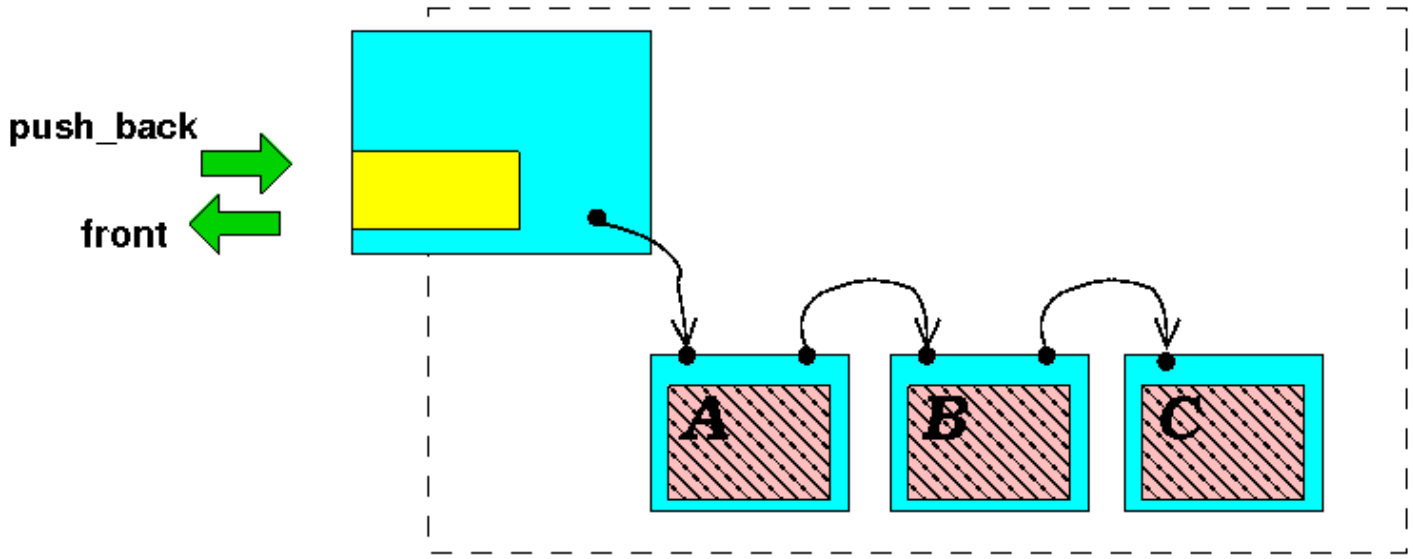
Lista obiektów `std::list<ObiektGraficzny>`

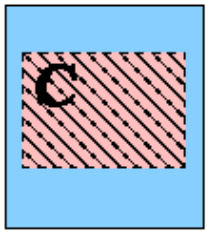


W liście można *próbować* umieścić obiekty klas pochodnych takich jak np. `Robot`, `Sciezka`, `Przeszkoda`.

Jednak do listy przekopiowane zostaną tylko podobiekty klasy bazowej.

Lista obiektów `std::list<ObiektGraficzny>`

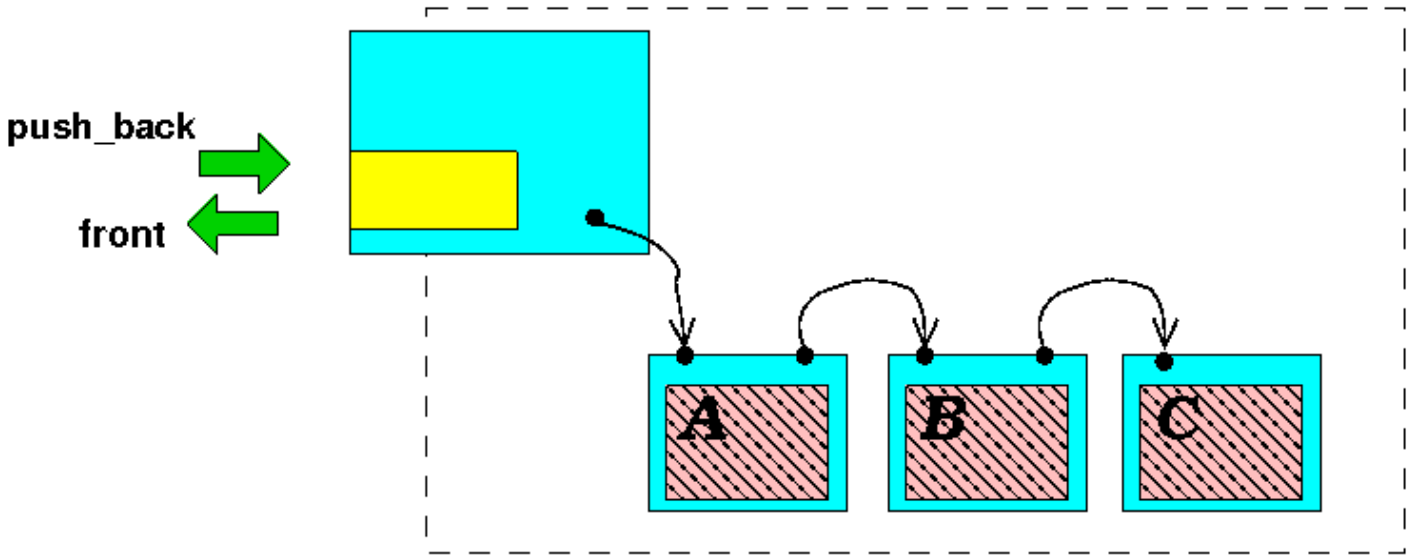




W liście można *próbować* umieścić obiekty klas pochodnych takich jak np. `Robot`, `Sciezka`, `Przeszkoda`.

Jednak do listy przekopiowane zostaną tylko podobiekty klasy bazowej.

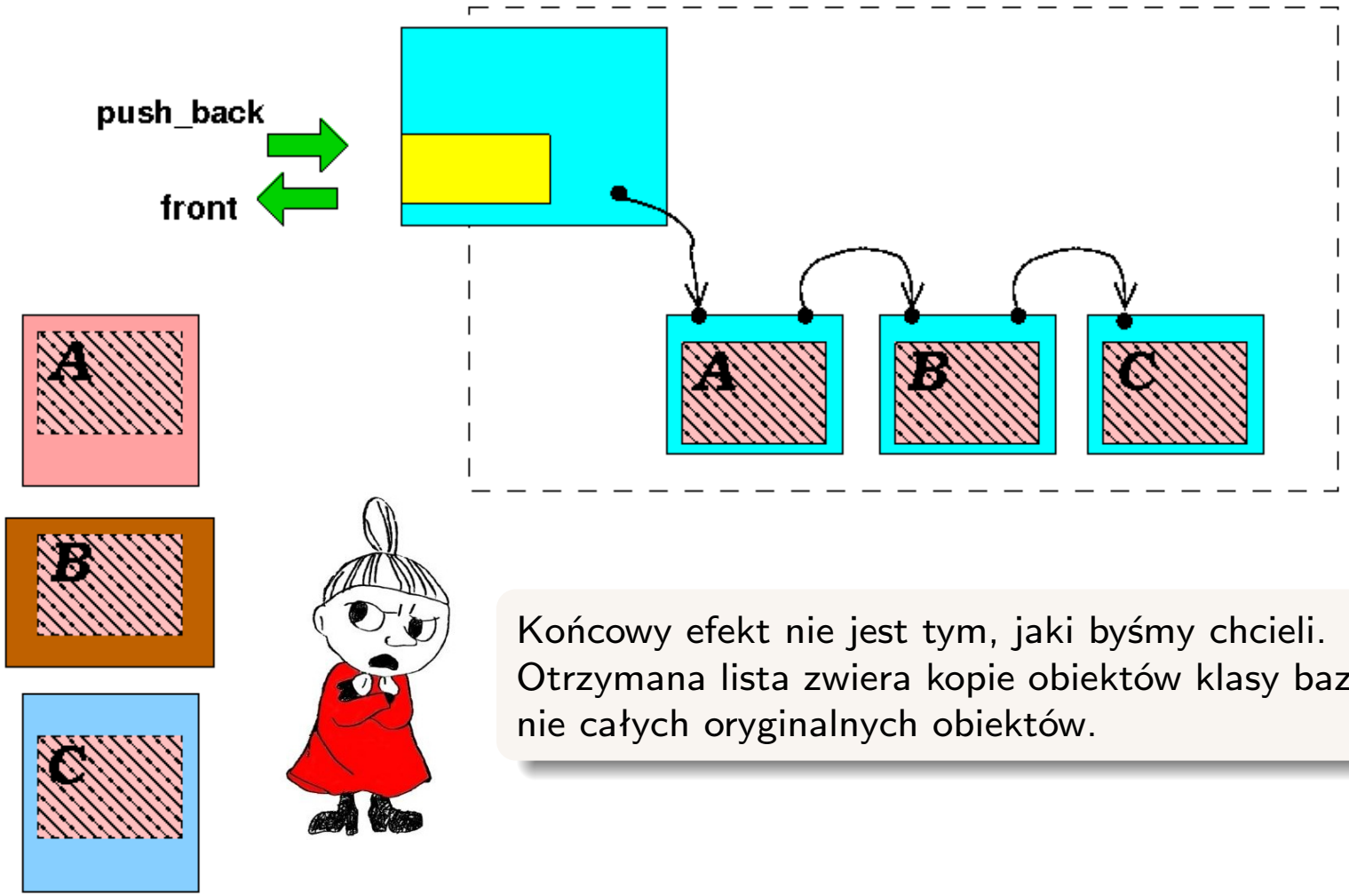
Lista obiektów `std::list<ObiektGraficzny>`



W liście można *próbować* umieścić obiekty klas pochodnych takich jak np. `Robot`, `Sciezka`, `Przeszkoda`.

Jednak do listy przekopiowane zostaną tylko podobiekty klasy bazowej.

Lista obiektów `std::list<ObiektGraficzny>`

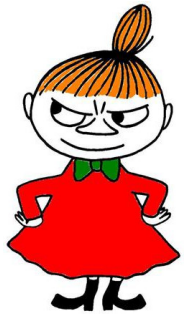


Końcowy efekt nie jest tym, jaki byśmy chcieli.
Otrzymana lista zwiera kopie obiektów klasy bazowej, a nie całych oryginalnych obiektów.

Plan prezentacji

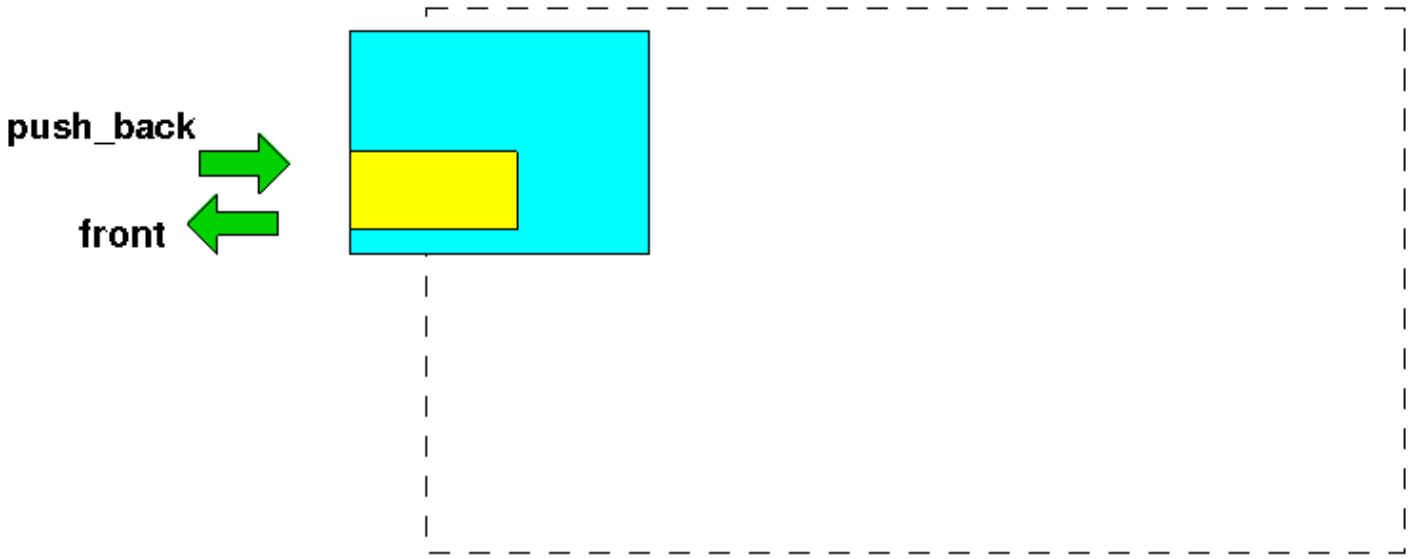
Lista wskaźników std::list<ObiektGraficzny*>

```
int main()
{
  std::list<ObiektGraficzny*>  Lst;

  cout << "== Przed ==========" << endl;
  Lst.push_back(new Robot());
  Lst.push_back(new Sciezka());
  Lst.push_back(new Przeszkoda());
  cout << "== Po ==========" << endl;
}
```



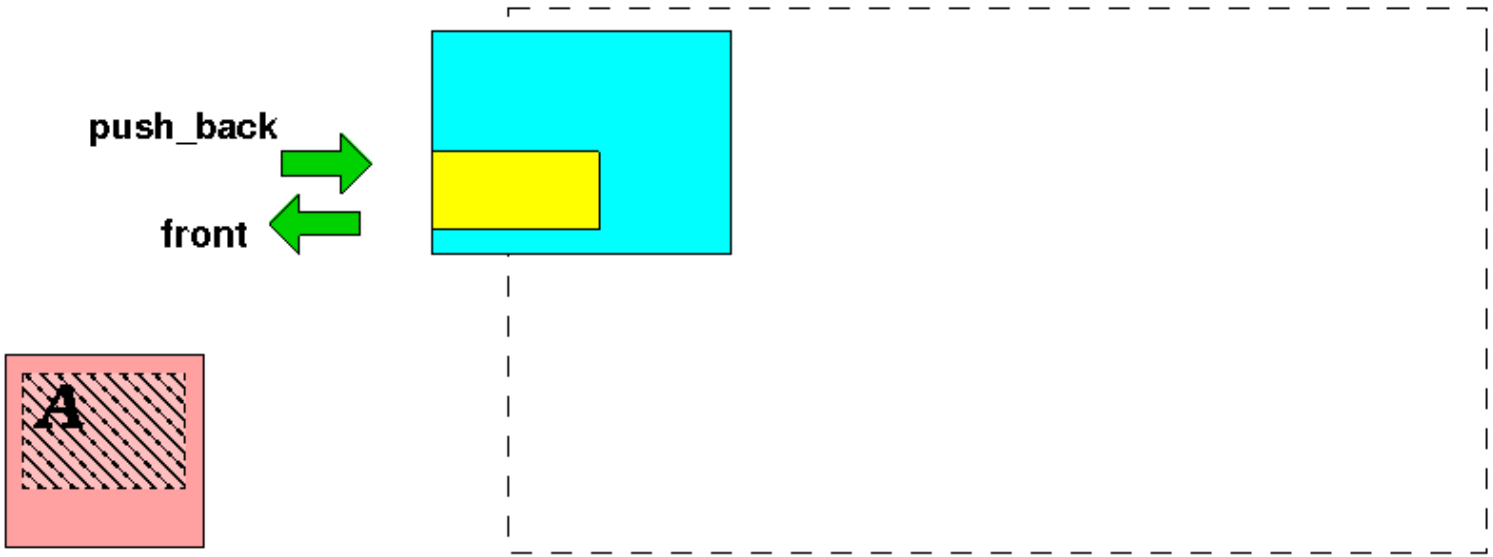
Dzięki dziedziczeniu zachodzić będzie niejawne rzutowanie na wskaźnik do klasy bazowej. Tak więc niniejszy kod zostanie zaakceptowany przez kompilator. Tym razem na liście znajdą się same wskaźniki.

Lista wskaźników – jak to działą



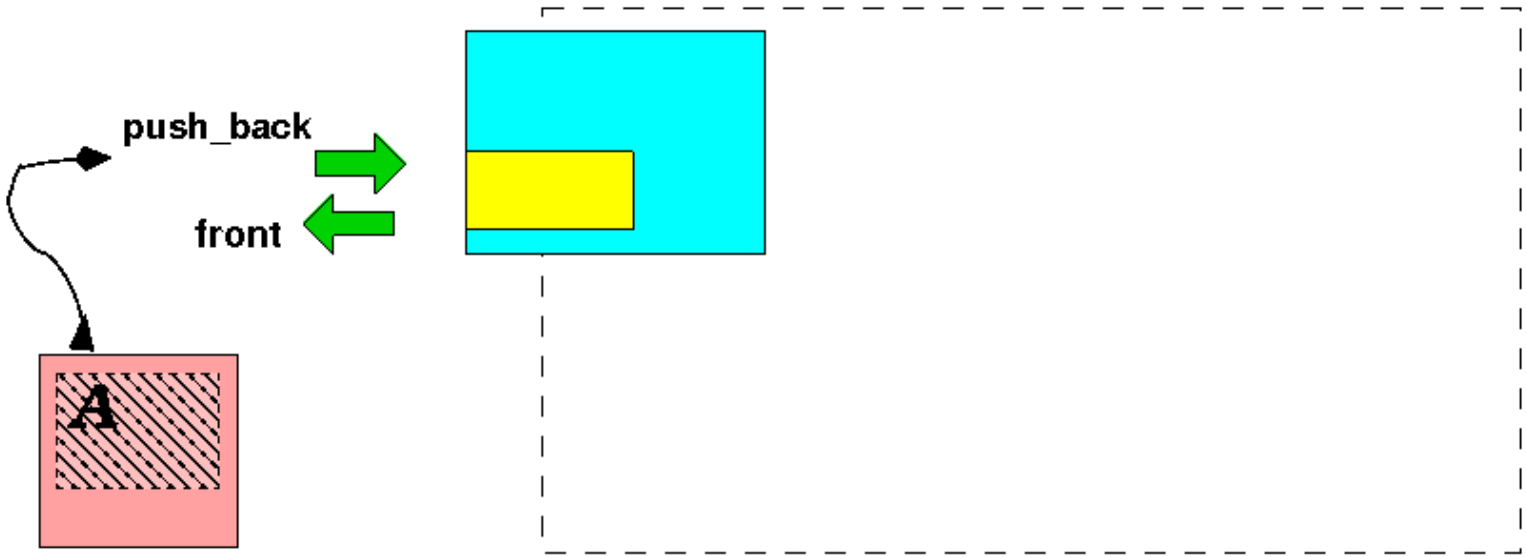
Zamiast całych obiektów na liście możemy umieścić jedynie wskaźniki na obiekty. Dzięki dziedziczeniu i niejawnemu rzutowaniu *w górę* możemy jako argumentów użyć wskaźników na obiekty klasy pochodnej.

Lista wskaźników – jak to działą



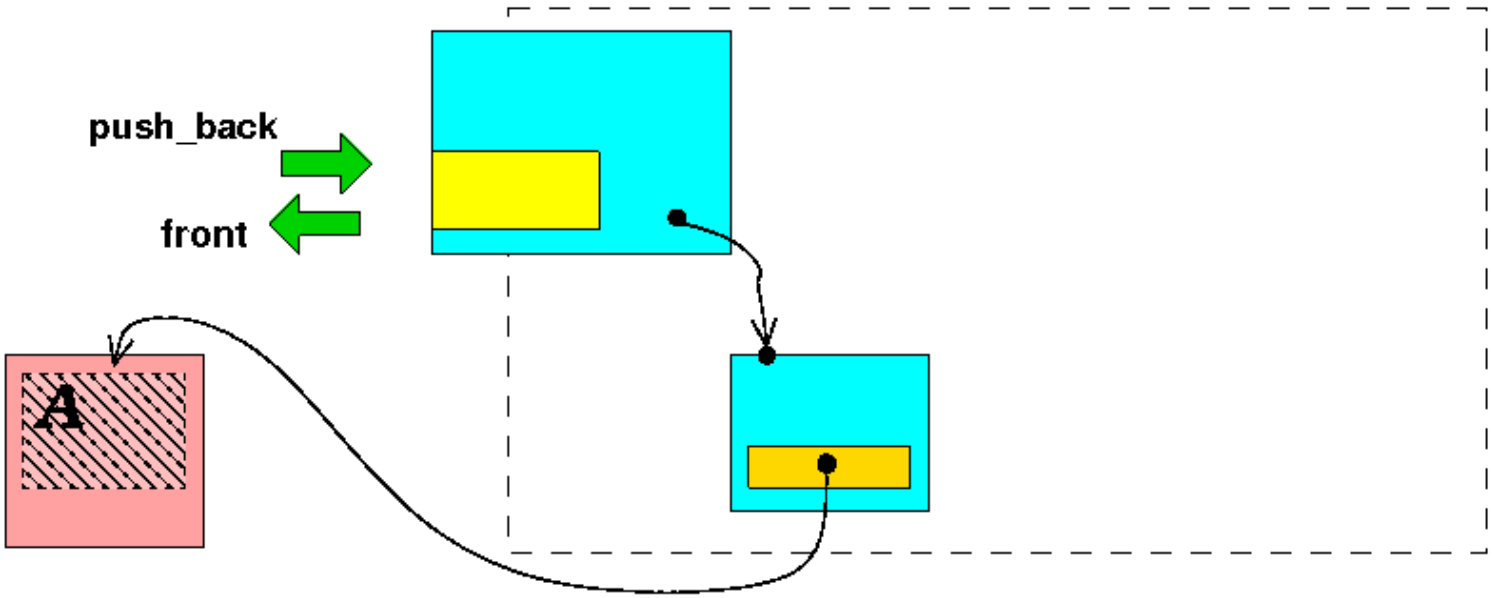
Zamiast całych obiektów na liście możemy umieścić jedynie wskaźniki na obiekty. Dzięki dziedziczeniu i niejawnemu rzutowaniu *w górę* możemy jako argumentów użyć wskaźników na obiekty klasy pochodnej.

Lista wskaźników – jak to działą



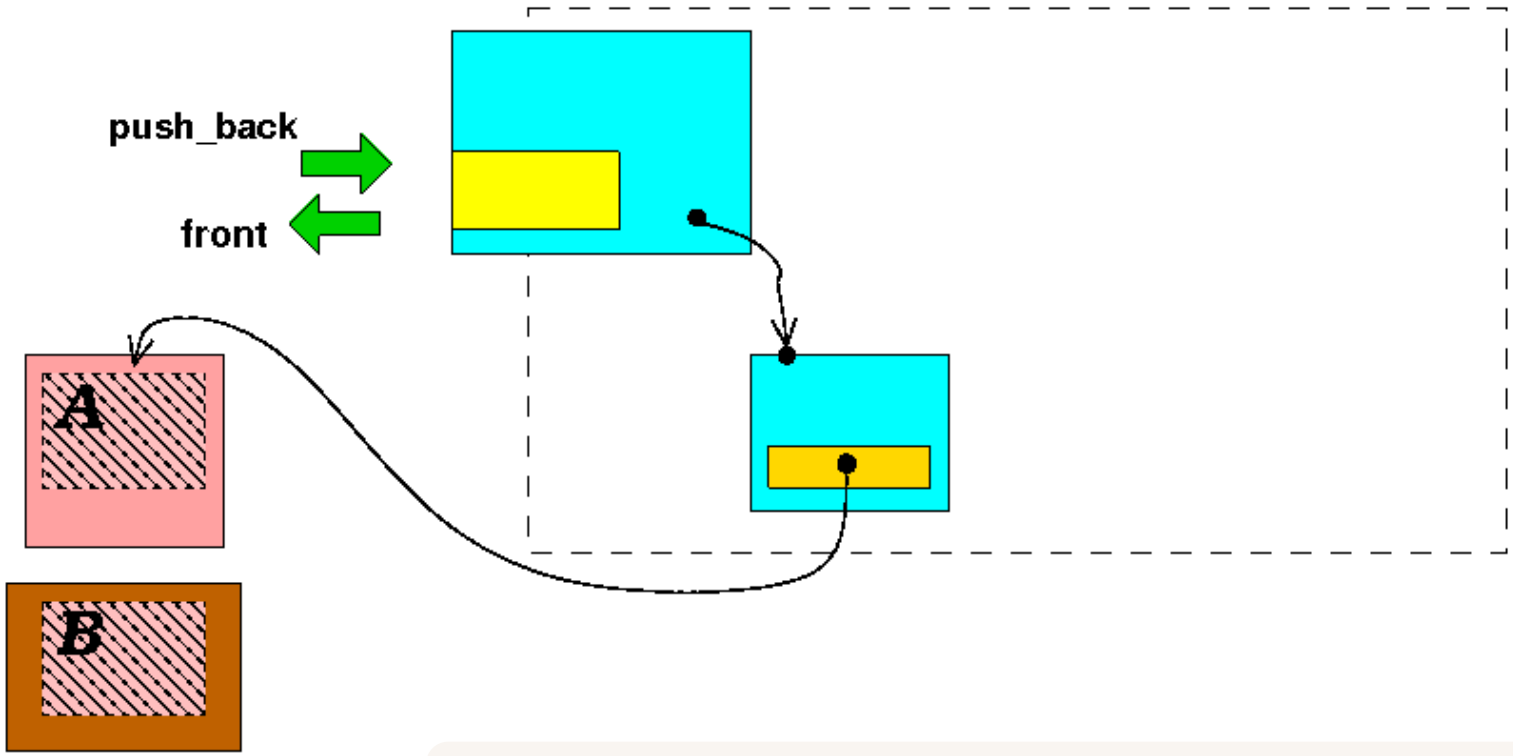
Zamiast całych obiektów na liście możemy umieścić jedynie wskaźniki na obiekty. Dzięki dziedziczeniu i niejawnemu rzutowaniu *w górę* możemy jako argumentów użyć wskaźników na obiekty klasy pochodnej.

Lista wskaźników – jak to działą



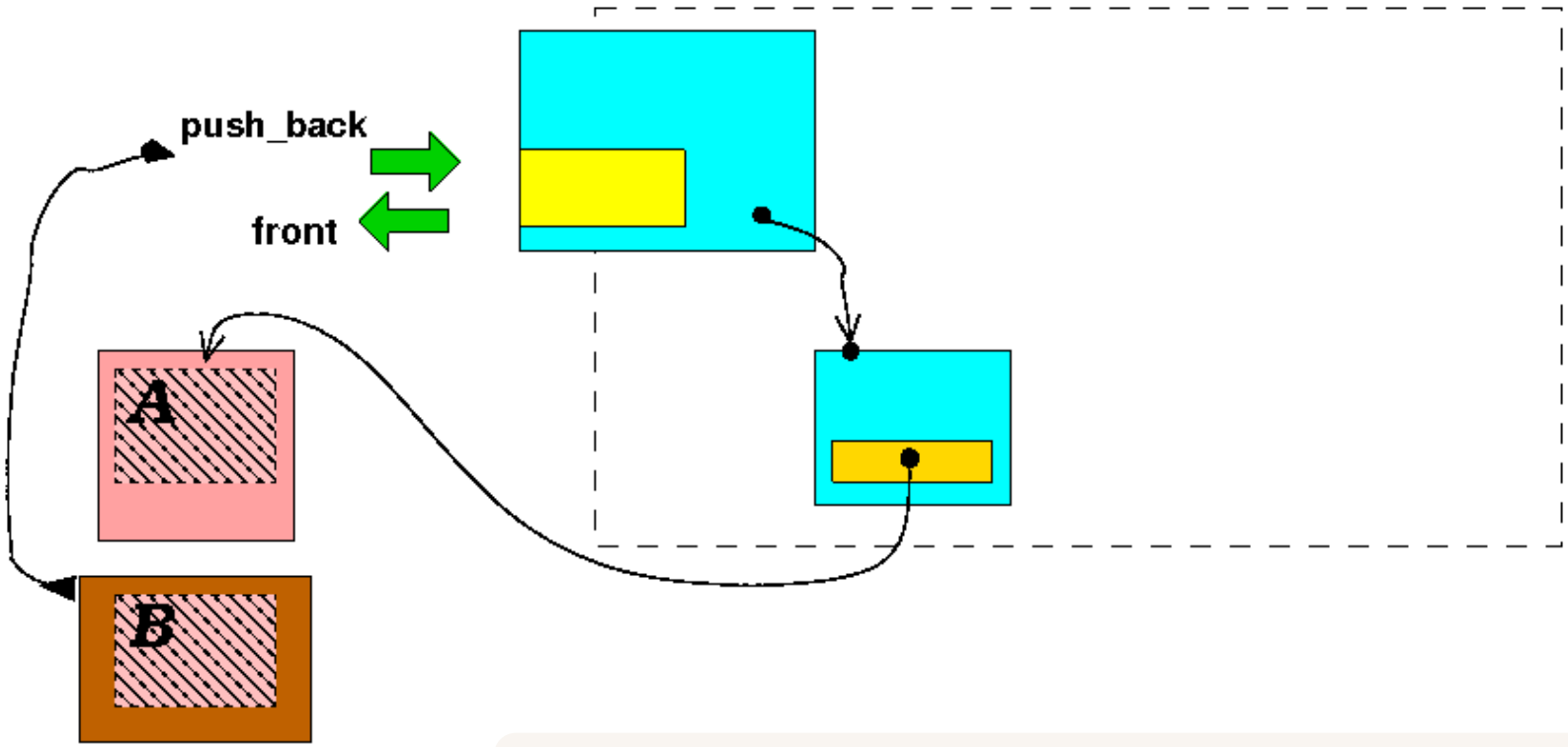
Zamiast całych obiektów na liście możemy umieścić jedynie wskaźniki na obiekty. Dzięki dziedziczeniu i niejawnemu rzutowaniu *w górę* możemy jako argumentów użyć wskaźników na obiekty klasy pochodnej.

Lista wskaźników – jak to działą



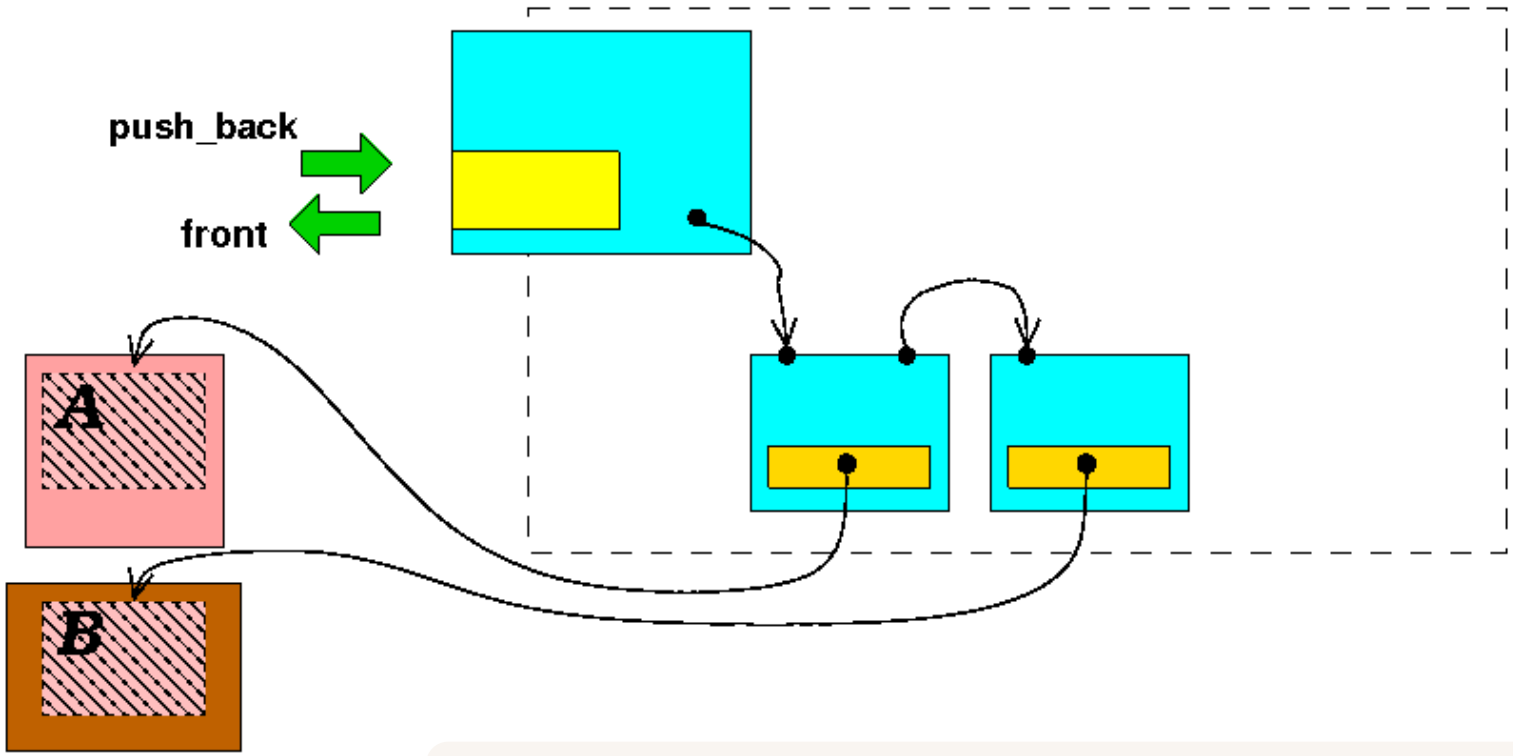
Zamiast całych obiektów na liście możemy umieścić jedynie wskaźniki na obiekty. Dzięki dziedziczeniu i niejawnemu rzutowaniu *w górę* możemy jako argumentów użyć wskaźników na obiekty klasy pochodnej.

Lista wskaźników – jak to działą



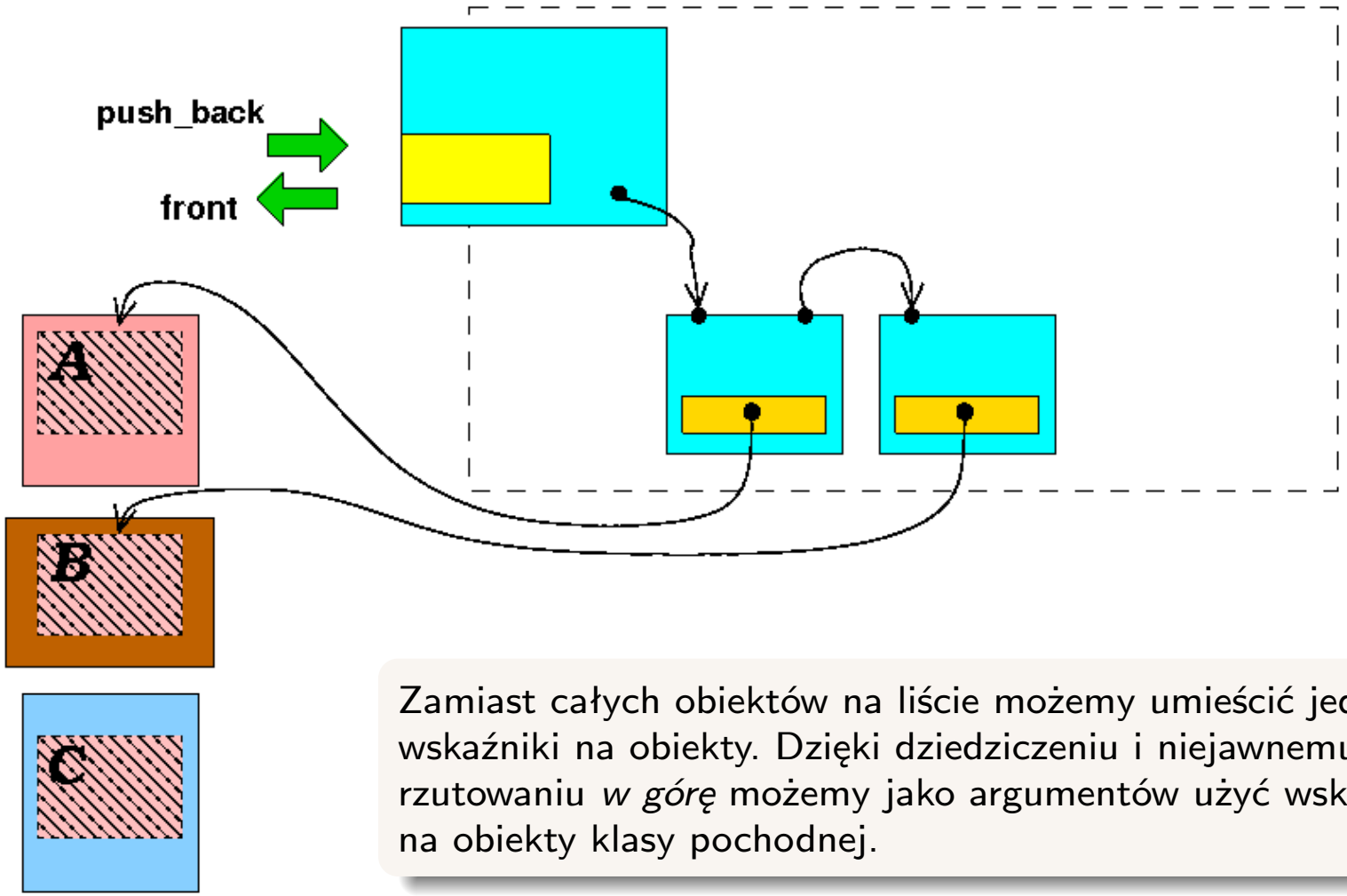
Zamiast całych obiektów na liście możemy umieścić jedynie wskaźniki na obiekty. Dzięki dziedziczeniu i niejawnemu rzutowaniu *w górę* możemy jako argumentów użyć wskaźników na obiekty klasy pochodnej.

Lista wskaźników – jak to działą



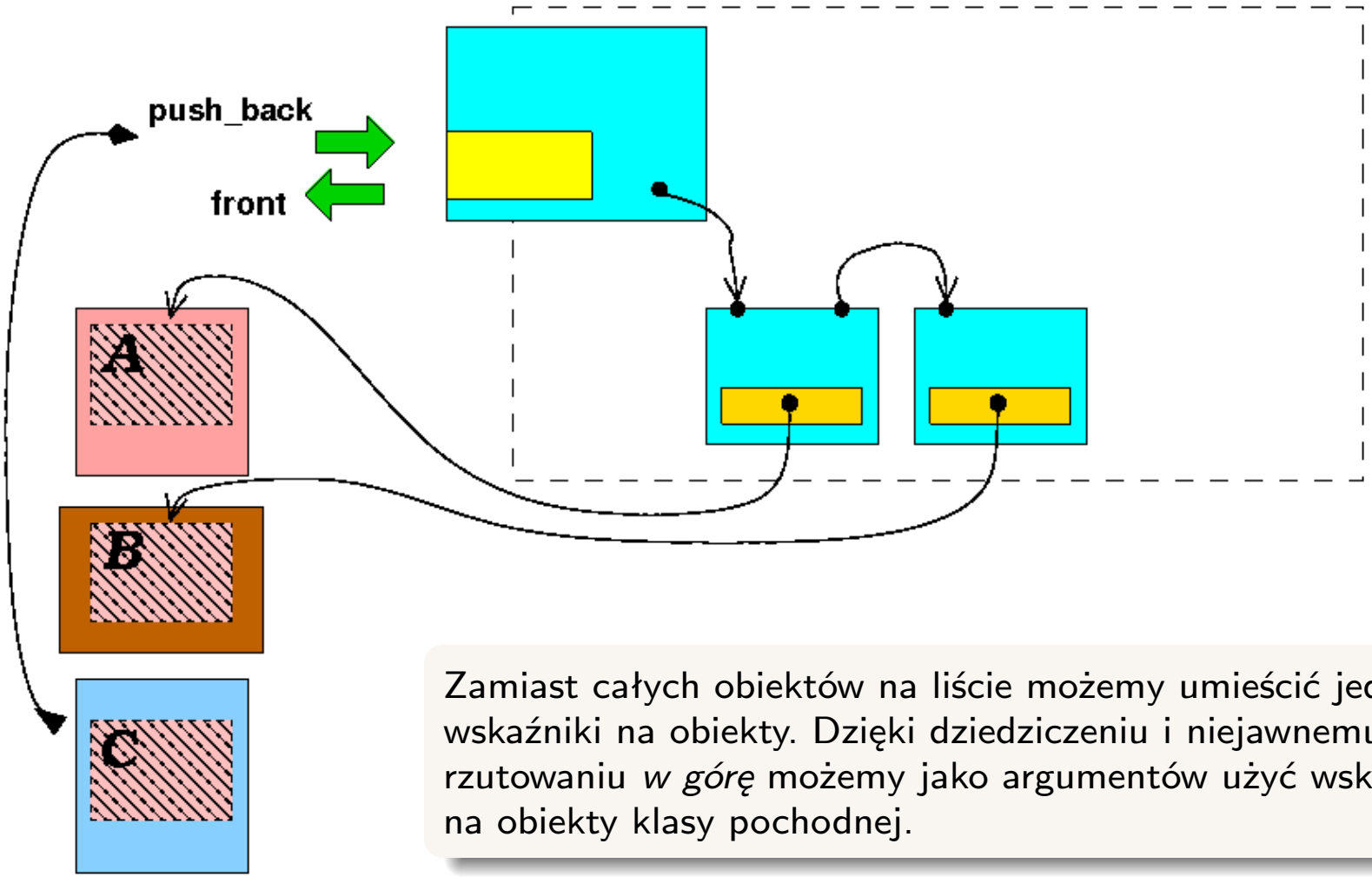
Zamiast całych obiektów na liście możemy umieścić jedynie wskaźniki na obiekty. Dzięki dziedziczeniu i niejawnemu rzutowaniu *w górę* możemy jako argumentów użyć wskaźników na obiekty klasy pochodnej.

Lista wskaźników – jak to działą



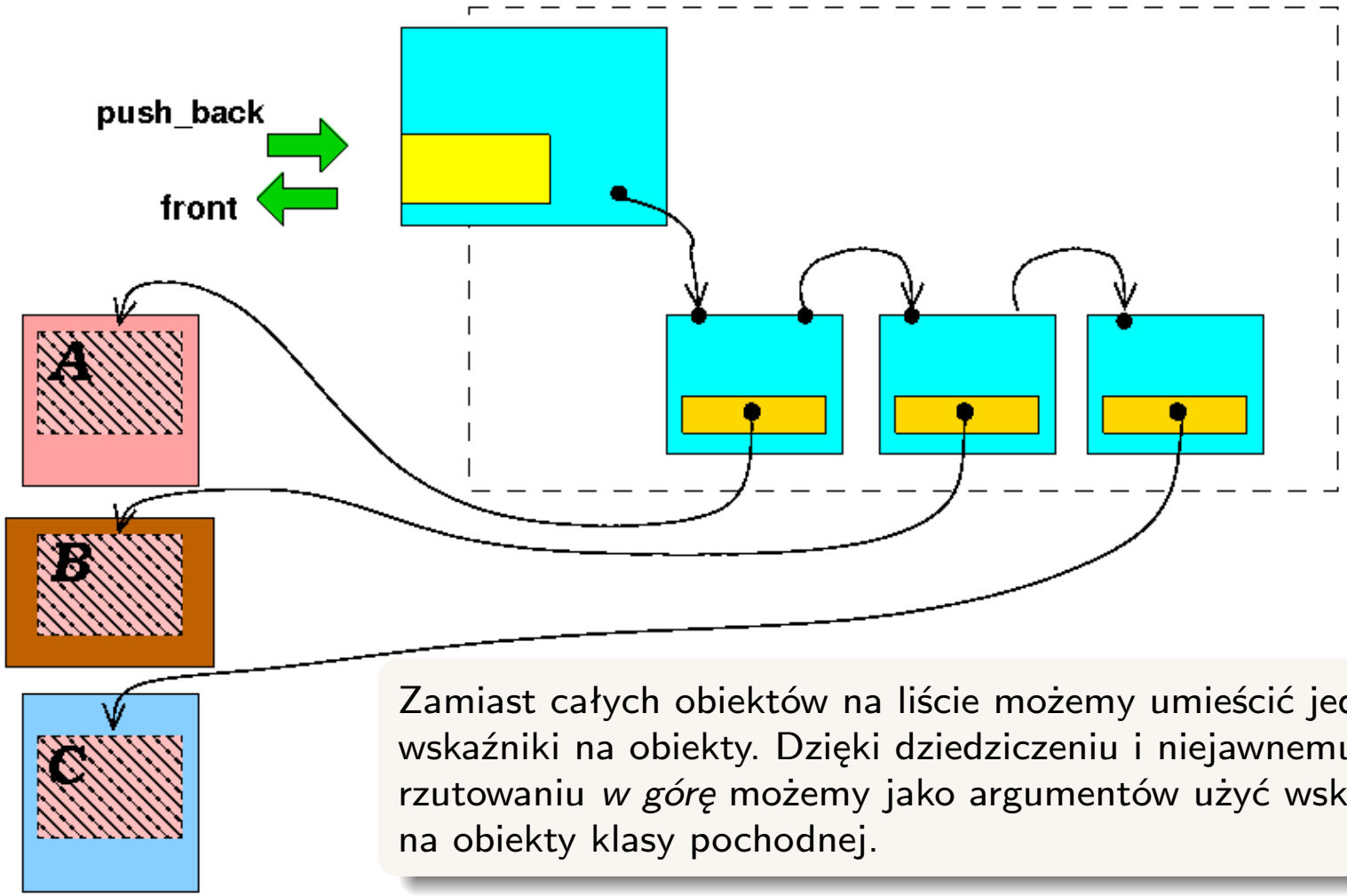
Zamiast całych obiektów na liście możemy umieścić jedynie wskaźniki na obiekty. Dzięki dziedziczeniu i niejawnemu rzutowaniu *w górę* możemy jako argumentów użyć wskaźników na obiekty klasy pochodnej.

Lista wskaźników – jak to działą



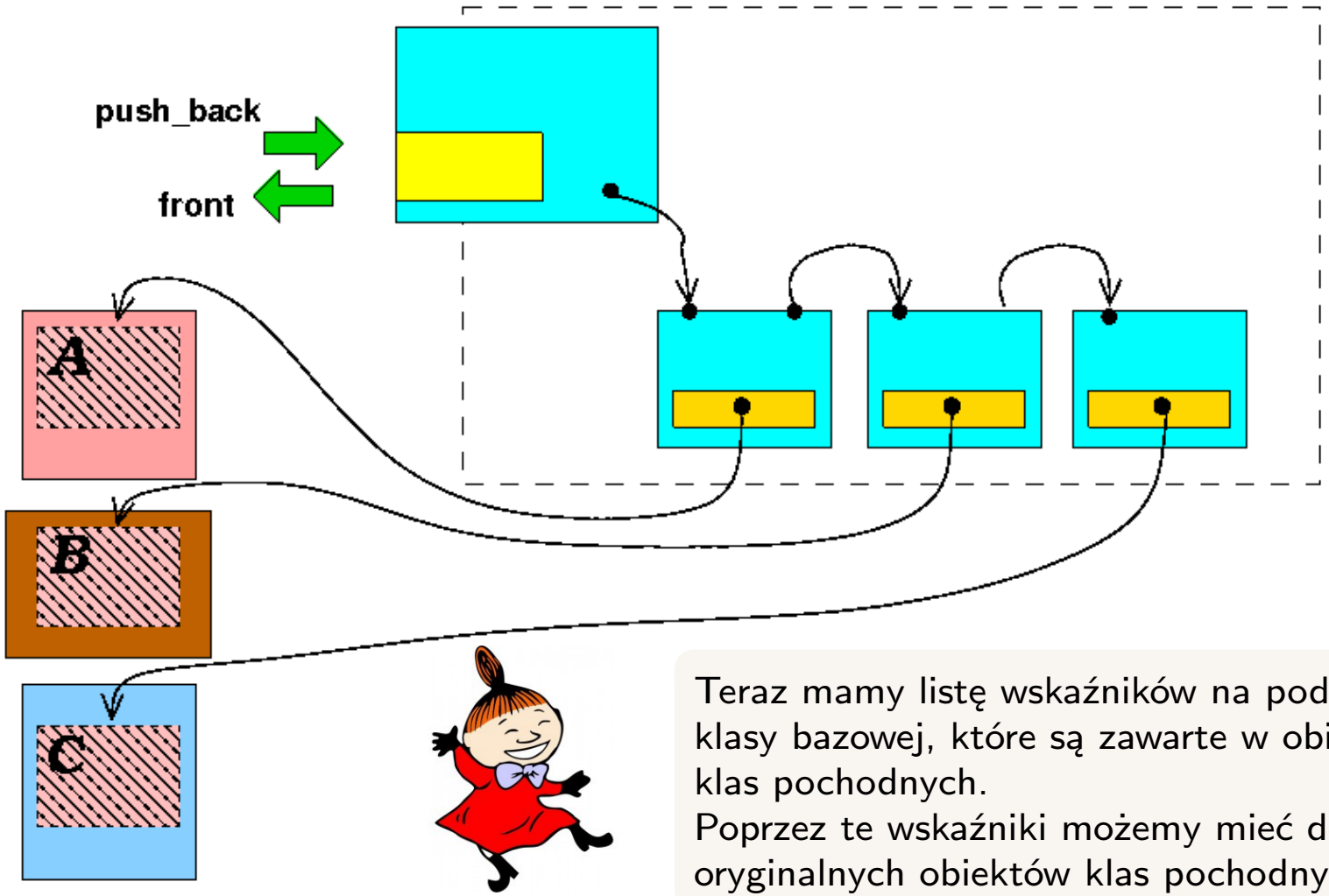
Zamiast całych obiektów na liście możemy umieścić jedynie wskaźniki na obiekty. Dzięki dziedziczeniu i niejawnemu rzutowaniu *w górę* możemy jako argumentów użyć wskaźników na obiekty klasy pochodnej.

Lista wskaźników – jak to działą



Zamiast całych obiektów na liście możemy umieścić jedynie wskaźniki na obiekty. Dzięki dziedziczeniu i niejawnemu rzutowaniu *w górę* możemy jako argumentów użyć wskaźników na obiekty klasy pochodnej.

Lista wskaźników – jak to działą



Teraz mamy listę wskaźników na podobiekty klasy bazowej, które są zawarte w obiektach klas pochodnych.
Poprzez te wskaźniki możemy mieć dostęp do oryginalnych obiektów klas pochodnych.

Hierarchia dziedziczenia – metody wirtualne

```
class ObiektGraficzny {
 public:
  virtual const char* NazwaTypu() const {return "ObiektGraficzny";}
};

class Robot: public: ObiektGraficzny {
 public:
  virtual const char* NazwaTypu() const { return "Robot"; }
};

class Sciezka: public: ObiektGraficzny {
 public:
  virtual const char* NazwaTypu() const { return "Sciezka"; }
};

class Przeszkoda: public: ObiektGraficzny {
 public:
  virtual const char* NazwaTypu() const { return "Przeszkoda"; }
};
```

Iteracja listy std::list<ObiektGraficzny*>

```
int main()
{
 std::list<ObiektGraficzny*>  Lst;

 Lst.push_back(new Robot());
 Lst.push_back(new Sciezka());
 Lst.push_back(new Przeszkoda());

 for (const ObiektGraficzny *wObGraf : Lst) {

   cout << "Typ: " << wObGraf->NazwaTypu() << endl;

 }
}
```

Wykorzystując hierarchię dziedziczenia oraz wprowadzoną metodę wirtualną, mając wskaźnik na obiekt klasy bazowej możemy *uruchomić* kod metody wirtualnej, który jest redefiowany w klasie pochodnej.

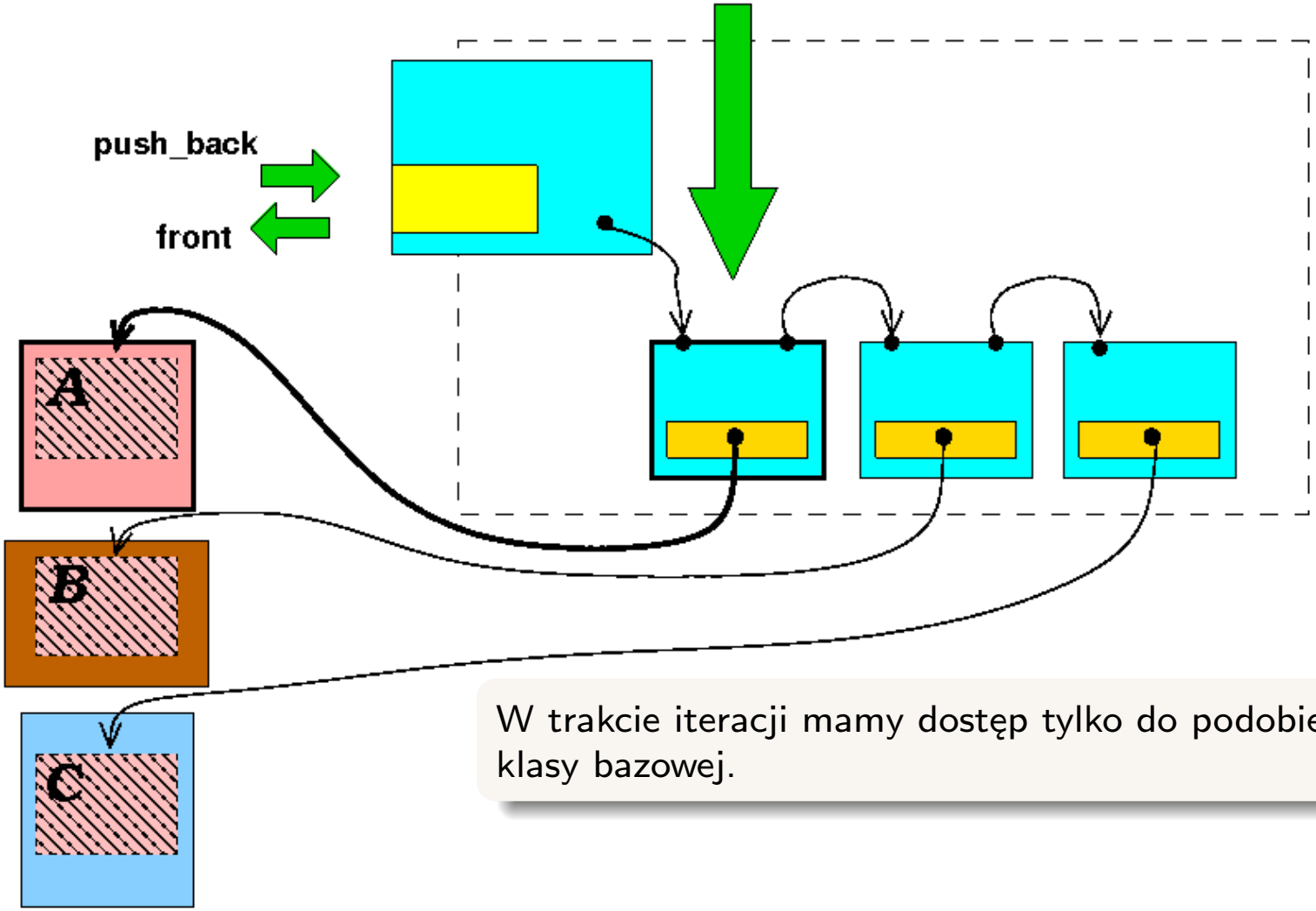
Iteracja listy std::list<ObiektGraficzny*>

```
int main()
{
 std::list<ObiektGraficzny*>  Lst;

 Lst.push_back(new Robot());
 Lst.push_back(new Sciezka());
 Lst.push_back(new Przeszkoda());

 for (const ObiektGraficzny *wObGraf : Lst) {

   cout << "Typ: " << wObGraf->NazwaTypu() << endl;

 }
}
```

Wynik działania:

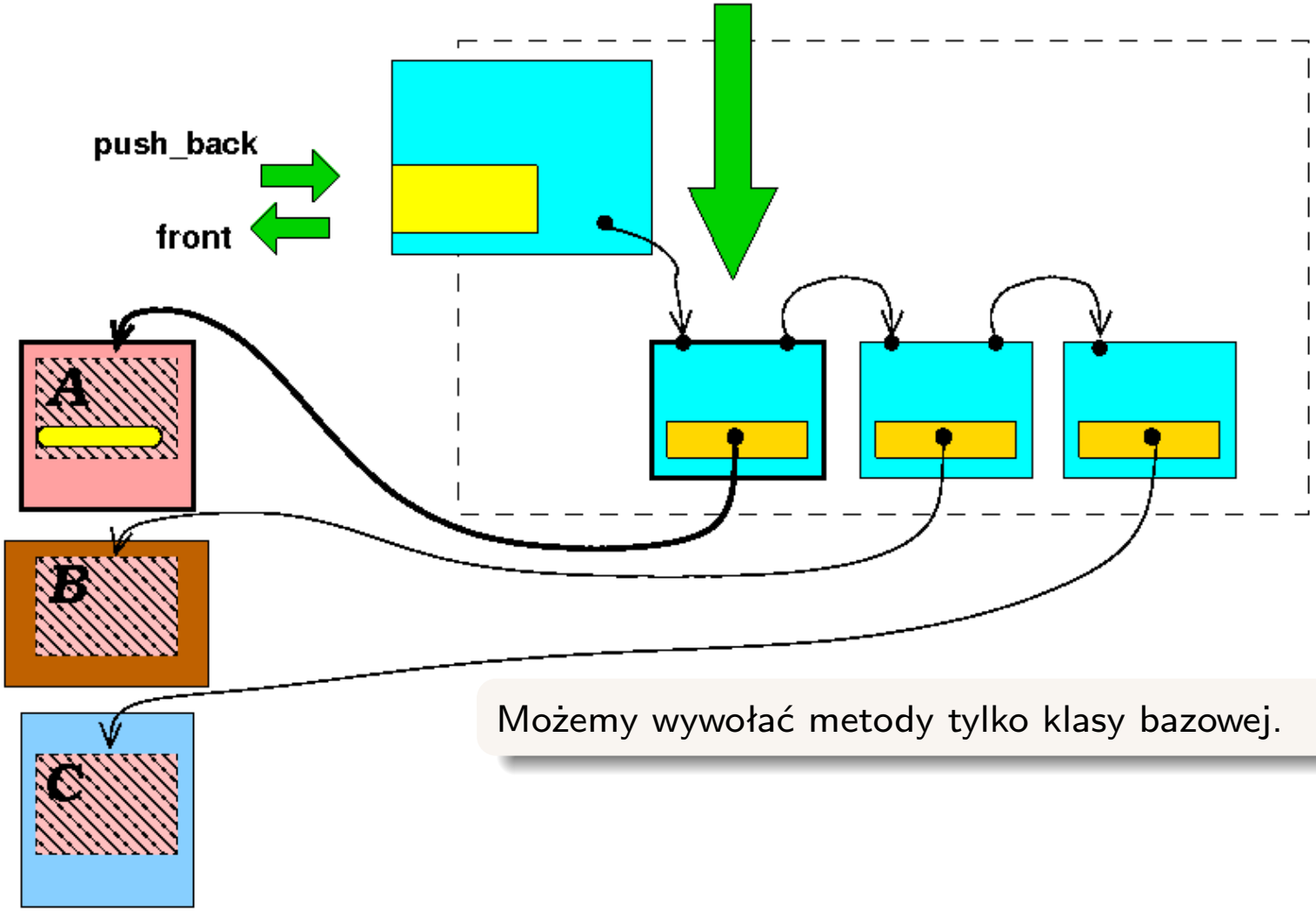
```
Typ: Robot
Typ: Sciezka
Typ: Przeszkoda
```

Iteracja z wywoływaniem metody wirtualnej



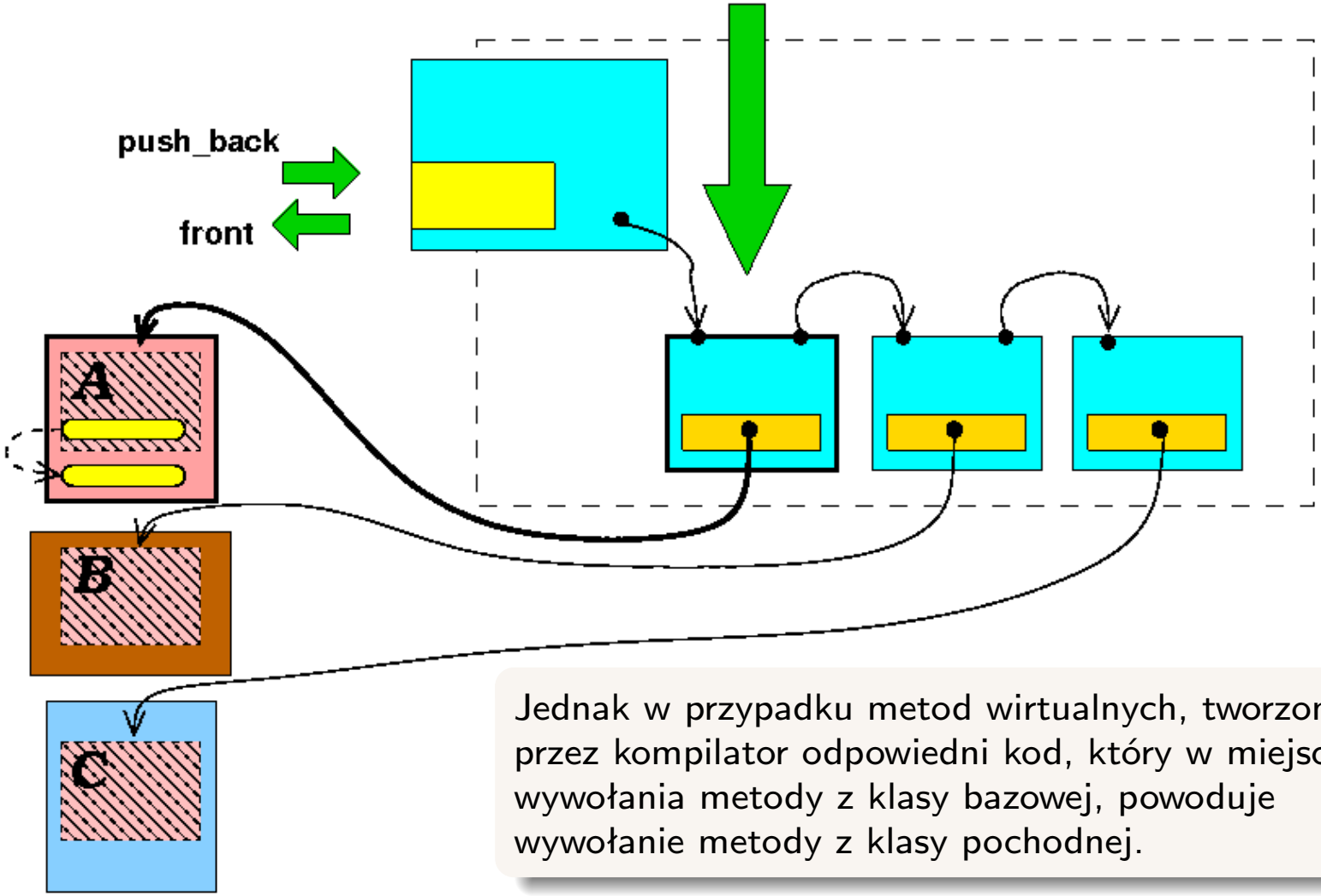
W trakcie iteracji mamy dostęp tylko do podobiektów klasy bazowej.

Iteracja z wywoływaniem metody wirtualnej



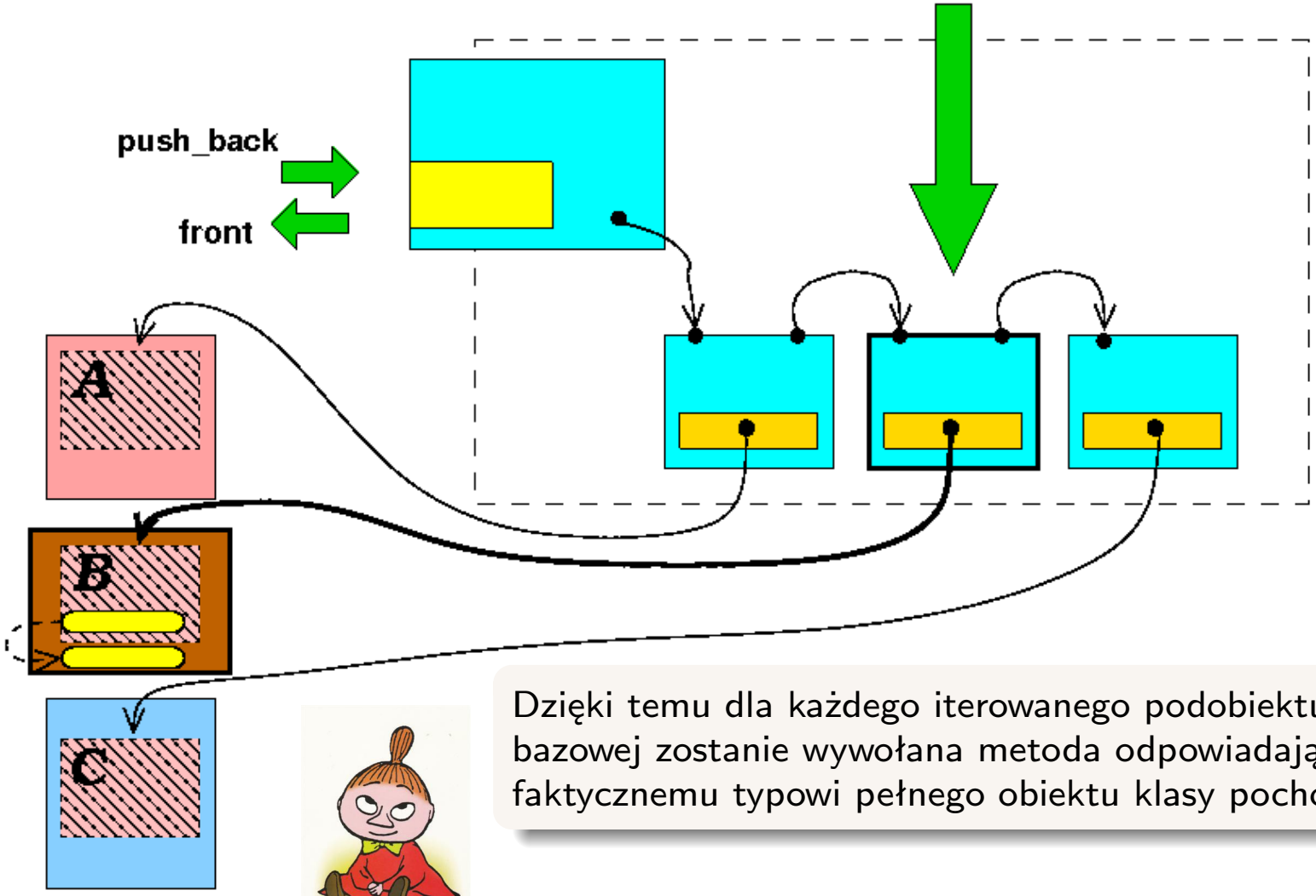
Możemy wywołać metody tylko klasy bazowej.

Iteracja z wywoływaniem metody wirtualnej



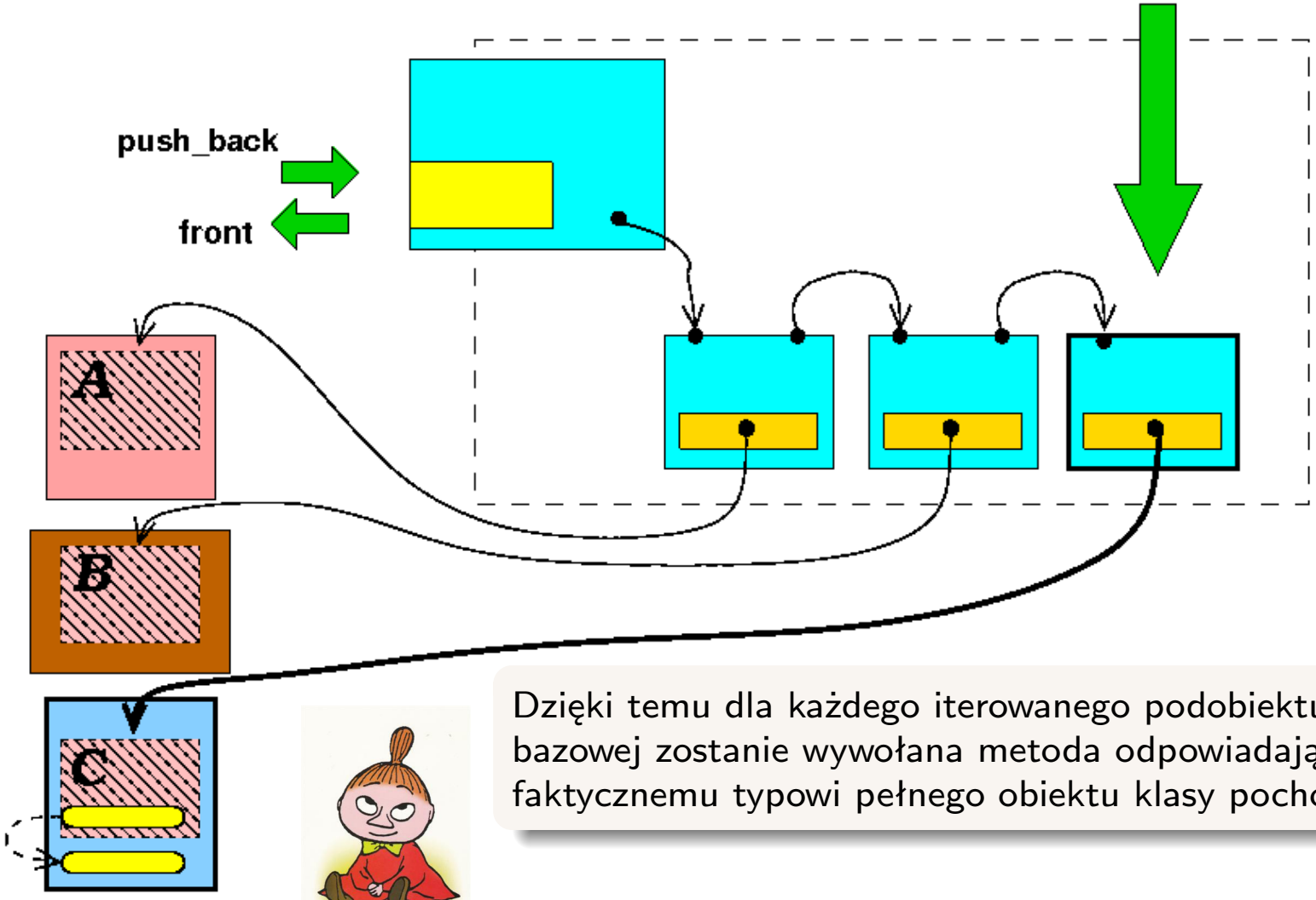
Jednak w przypadku metod wirtualnych, tworzony jest przez kompilator odpowiedni kod, który w miejsce wywołania metody z klasy bazowej, powoduje wywołanie metody z klasy pochodnej.

Iteracja z wywoływaniem metody wirtualnej



Dzięki temu dla każdego iterowanego podobiektu klasy bazowej zostanie wywołana metoda odpowiadająca faktycznemu typowi pełnego obiektu klasy pochodnej.

Iteracja z wywoływaniem metody wirtualnej



Dzięki temu dla każdego iterowanego podobiektu klasy bazowej zostanie wywołana metoda odpowiadająca faktycznemu typowi pełnego obiektu klasy pochodnej.

Koniec prezentacji
Dziękuję za uwagę