Podstawy programowania obiektowego

dr inż. Piotr Szwed
Katedra Informatyki Stosowanej
C2, pok. 403

e-mail: pszwed@agh.edu.pl

http://home.agh.edu.pl/~pszwed/

Aktualizacja: 16.05.2020

10. Semantyka przenoszenia

Wędrówka danych

Klasa do śledzenia operacji kopiowania i przypisania

```
class A{
    string ctx;
public:
    A(const char*_ctx):ctx(_ctx){
        cout<<"Ctor ["<<ctx<<"] "<<hex<<this<<endl;
    }
    A(const A&other){
        ctx = other.ctx+"_copy_of("+to_text(&other)+")";
        cout<<"Ctor ["<<ctx<<"] "<<hex<<this<<endl;
    }
    A&operator=(const A&other){
        ctx = other.ctx+"_assignment_of("+to_text(&other)+")";
        cout<<"assign ["<<ctx<<"] "<<hex<<this<<endl;
        return *this;
    }
    static string to_text(const void*ptr){
        ostringstream s;
        s<<hex<<ptr;
        return s.str();
    }

};
```

Wędrówka danych (C++98)

```
void f_in(A a){

}

A f_out(){
    A a("in_f_out");
    return a;
}

int main(){
    A a("a in_main");
    f_in(a);
    cout<<endl;
    A a2 = f_out();
    cout<<endl;
    A a3("a3 in main");
    a3 = a2;
}
```

Obiekt przekazany przez wartość

Zwracany obiekt. Obiekt tworzony wewnątrz funkcji. Następnie a jest kopiowany do obiektu tymczasowego

Do funkcji przekazana jest kopia `a("a in_main")`

Obiekt a2 inicjowany wartością obiektu tymczasowego (konstruktor kopiujący)

Obiekt a2 inicjowany wartością obiektu tymczasowego (konstruktor kopiujący)

Wędrówka danych (C++98)

```
void f_in(A a){

}

A f_out(){
    A a("in_f_out");
    return a;
}

int main(){
    A a("a in_main");
    f_in(a);
    cout<<endl;
    A a2 = f_out();
    cout<<endl;
    A a3("a3 in main");
    a3 = a2;
}
```

Konstruktor A `a("a in_main")` i wywołanie `f_in()`

```
Ctor [a in_main] 0xffffcbf8
Ctor [a in_main_copy_of(0xffffcbf8)]
0xffffcc00
```

Utworzenie A `a("a in_f_out")`:

```
Ctor [in_f_out] 0xffffcb98
```

Kopiowanie do obiektu tymczasowego:

```
Ctor [in_f_out_copy_of(0xffffcb98)]
0xffffcc08
```

Kopiowanie z obiektu tymczasowego do `a2`:

```
Ctor
[in_f_out_copy_of(0xffffcb98)_copy_of(0xff
ffcc08)] 0xffffcbf0
```

Konstruktor A `a3("a3 in_main")`:

```
Ctor [a3 in main] 0xffffcbe8
```

Przypisanie `a2`:

```
assign
[in_f_out_copy_of(0xffffcb98)_copy_of(0xff
ffcc08)_assignment_of(0xffffcbf0)]
0xffffcbe8
```

Wędrówka danych (C++98)

```
void f_in(A a){

}

A f_out(){
    A a("in_f_out");
    return a;
}

int main(){
    A a("a in_main");
    f_in(a);
    cout<<endl;
    A a2 = f_out();
    cout<<endl;
    A a3("a3 in main");
    a3 = a2;
}
```

Wyobraźmy sobie, że nasz kontener ma na przykład 10000 elementów:

- Wywołując `f_in()` kopiujemy całą zawartość. Przy wyjściu z `f_in()` pamięć zostanie usunięta.
  Tego zawsze należy unikać!

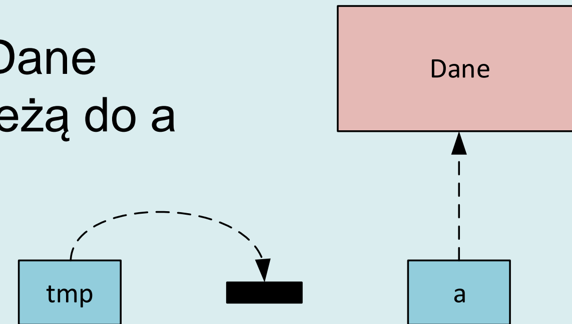
- Wywołanie `A a2=f_out()` to dwie operacje kopiowania i usuwania:
  1. a $\rightarrow$ obiekt tymczasowy
  2. Zawartość a jest usuwana przy wyjściu z funkcji
  3. obiekt tymczasowy $\rightarrow$ a2
  4. Zawartość obiektu tymczasowego jest usuwania po wykorzystaniu

  Bardzo nieefektywne!

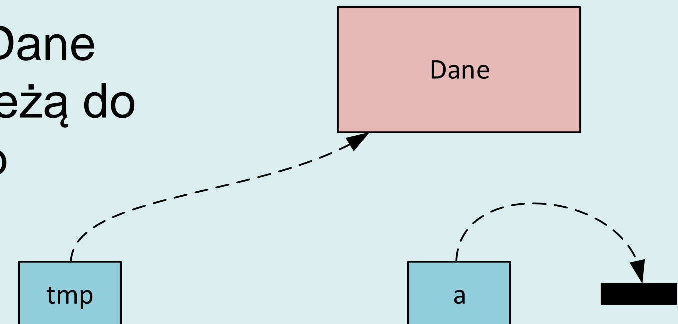
Przenoszenie - idea

W standardzie C++11 wprowadzono nowy mechanizm – przenoszenie zamiast kopiowania i usuwania.

1. Dane należą do a



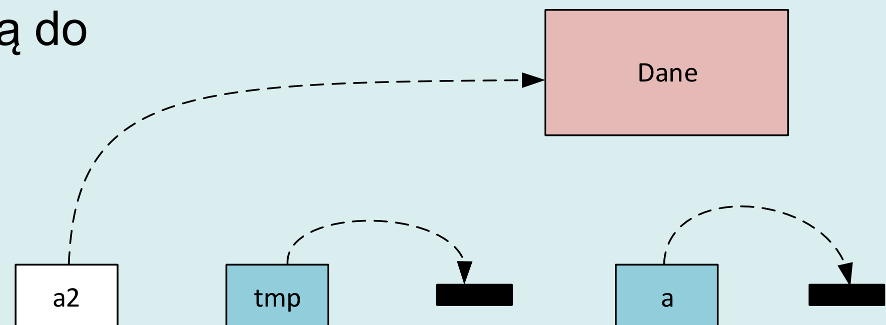
2. Dane należą do tmp




```
A f_out(){
    A a("in_f_out");
    return a;
}

A a2 = f_out();
```

3. Dane należą do a2



Przenoszenie

- Przenoszenie nie jest rozwiązaniem nowym. Zawsze było wykorzystywane w programach w C

```
char*doklej(char**dane){
    char*dane2=*dane;
    *dane=0;
    strcat(dane2," i psa");
    return dane2;
}

int main(){
    char*ptr=new char[256];
    strcpy(ptr,"Ala ma");
    char*ptr2=doklej(&ptr);
    cout<<(ptr== nullptr?"null":ptr)<<endl;
    cout<<ptr2<<endl;
    delete []ptr2;
}
```

- Celem mechanizmu jest uzyskanie takiej wydajności jak w języku C przy równoczesnym uniknięciu manipulacji wskaźnikami.
- Nowy typ danych: referencja prawostronna (rvalue reference)

Wartości lewo- i prawostronne

- W językach C/C++ wprowadza się terminy wartości lewostronnej (lvalue) i prawostronnej (rvalue).
- Definicja: `lvalue = rvalue;`
  - `lvalue` to wyrażenie, które może pojawić się po lewej stronie operatora przypisania – czyli np. zmienna, referencja zmiennej, wskaźnik wskazujący obiekt po dereferencji
  - `rvalue`, to wyrażenie, które może pojawić się po prawej stronie – np. zmienna, referencja, stała, adres zmiennej

`int a=12,b=7;`	a i b to lvalue, stałe to rvalue
`a*b`	rvalue,nie można wykonać `a*b=15;`
`int f1();` `f1()=15`	`f1()` to rvalue
`int*f2();` `*f2()=12;`	`*f2()` to lvalue
`int*ptr=&f1();`	Nie można pobrać adresu rvalue

Rvalue reference

Dla danego typu T:

- Zwykła (lewostronna) referencja zdeklarowana jest jako `T&ref`
- Referencja prawostronna jako `T&&ref`

```
int suma(int a, int b){
    return a+b;
}

int main() {
    int x = 1;
    int &lr1 = x;
    int tab[3] = {1, 2, 3};
    int &lr2 = tab[1];
    int &lr3 = suma(2, 4); // error: cannot bind non-const
      //lvalue reference of type 'int&' to an rvalue of type 'int'
…
```

Referencje lewostrone (bez modyfikatorów) możemy ustawiać wyłącznie na lvalue.

Rvalue reference

```
int main() {
 …
    const int &lr4 = 1;
    const int &lr4 = suma(2, 4);
    int &&rr1 = x; // error: cannot bind rvalue reference
                   //of type 'int&&' to lvalue of type 'int'
    int &&rr11 = 1;
    int &&rr2 = tab[1]; //error: cannot bind rvalue reference
                        //of type 'int&&' to lvalue of type 'int'
    int &&rr21 = tab[1] + 0;
    int &&rr3 = suma(2, 4);
    int&lr=rr3;
    int&&rr4=lr; //error: cannot bind rvalue reference
                 //of type 'int&&' to lvalue of type 'int'
}
```

- Referencje lewostronne typu `const` możemy ustawiać na rvalue. Automatycznie tworzone są obiekty tymczasowe inicjowane wartościami.
- Referencje prawostronne możemy ustawiać na rvalue, ale nie na lvalue.
- Referencjom lewostronnym można nadać wartość referencji prawostronnej, ale nie na odwrót.

Rvalue reference

- Możemy zaimplementować przeciążone wersje funkcji z parametrami typu lvalue reference i rvalue reference.
- Kompilator wybierze odpowiednią wersję w zależności od kontekstu wywołania.

```
int suma(int a, int b);{return a+b;}

void print(int&lvr){
    // wypisz adres i wartość obiektu
    cout<<hex<<&lvr<<dec<<" "<<lvr<<endl;
}

void print(int&&rvr){
    //wypisz wartość obiektu
    cout<<rvr<<endl;
}

int main(){
    int x = 1;
    print(x);                //lvalue
    print(suma(121,212)); //rvalue
}
```

Przykład

- Klasa `MyString` to niemodyfikowalny łańcuch znaków.
- `strdup()` to funkcja biblioteczna tworząca kopię tekstu na stercie (kompatypilna z `free()` ze `stdlib`).

```
class MyString{
public:
    char*ptr;
    MyString(const char*txt){
        ptr =strdup(txt);
        cout<<"Ctor create:"<<ptr<<endl;
    }
    ~MyString(){
        if(ptr){
            cout<<"Dtor delete:"<<ptr<<endl;
            ::free(ptr);
        }
    }
…
};
```

Przykład

```
class MyString{
public:
…
    MyString(const MyString&other){
        if(other.ptr){
            // copy - zrób kopię
            ptr=strdup(other.ptr);
            cout<<"Ctor copy:"<<ptr<<endl;
        }
    }
    MyString(MyString&&other){
        if(other.ptr){
            // move - przenieś
            ptr=other.ptr;
            other.ptr= nullptr;
            cout<<"Ctor move:"<<ptr<<endl;
        }
    }
};
```

Dodajemy dwie implementacje konstruktora kopiującego.

Pierwsza nie modyfikuje `other`, ale sporządza kopię jego zawartości.

Druga przenosi własność tekstu wskazywanego przez `other.ptr`.

Wskaźnik other.ptr jest zerowany. Destruktor `other` nie usunie tekstu.

Przykład

```
MyString ala_ma_kota(){
    cout<<"in ala_ma_kota\n";
    MyString s("Ala ma kota");
    return s;
}

void send(MyString s){
    cout<<"in send\n";
}

int main(){
    cout<<"in main 1\n";
    MyString s = ala_ma_kota();
    cout<<"in main 2\n";
    MyString s2("Ola ma kota");
    cout<<"in main 3\n";
    send(s2);
    cout<<"in main 4\n";
}
```

```
in main 1
in ala_ma_kota
Ctor create:Ala ma kota
Ctor move:Ala ma kota
Ctor move:Ala ma kota
in main 2
Ctor create:Ola ma kota
in main 3
Ctor copy:Ola ma kota
in send
Dtor delete:Ola ma kota
in main 4
Dtor delete:Ola ma kota
Dtor delete:Ala ma kota
```

Przykład

W funkcji `ala_ma_kota()` utworzono obiekt, dalej został on przeniesiony do obiektu tymczasowego i dalej przeniesiony do `s` w `main()`

```
cout<<"in main 1\n";
MyString s = ala_ma_kota();
```

```
in main 1
in ala_ma_kota
Ctor create:Ala ma kota
Ctor move:Ala ma kota
Ctor move:Ala ma kota
```

W `main()` utworzono obiekt `s2`. Podczas wywołania `send()` na stosie została umieszczona jego kopia. Przy wyjściu z `send()` został wywołany destruktor i usunął kopię.

```
cout<<"in main 2\n";
MyString s2("Ola ma kota");
cout<<"in main 3\n";
send(s2);
```

```
in main 2
Ctor create:Ola ma kota
in main 3
Ctor copy:Ola ma kota
in send
Dtor delete:Ola ma kota
```

```
in main 4
Dtor delete:Ola ma kota
Dtor delete:Ala ma kota
```

Obiekty `s` i `s2` zostały usunięte przy wyjściu z `main()`.

Zmiany w klasie Vector

```
class Vector{
    friend class VectorIterator;
protected:
    double*start;
    double*end;
    int capacity;
    void copy(const Vector&other);
    void free();
    void move(Vector&other);

    // ...
```

```
void Vector::move(Vector&other){
    this->start = other.start;
    this->end = other.end;
    this->capacity = other.capacity;
    other.start=other.end=0;
    other.capacity = 0;

}
```

Dodajemy trzecią pomocniczą funkcję: `move()`.
Czyli:

- `copy()` – kopiuje zawartość `other`, nie zmieniając jego danych
- `free()` – zwalania pamięć
- `move()` – przenosi zawartość `other`

Zmiany w klasie Vector

Dwa konstruktory kopiujące, tradycyjny `Vector(const Vector&other)` oraz implementujący semantykę przenoszenia `Vector(Vector&&other)`.

```
Vector(const Vector&other){ copy(other); }
Vector(Vector&&other){ move(other); }
```

```
Vector&Vector::operator=(const Vector&other){
    if(&other!=this){
        free();
        copy(other);
    }
    return *this;
}

Vector&Vector::operator=(Vector&&other){
    if(&other!=this){
        free();
        move(other);
    }
    return *this;
}
```

Analogicznie, dwa operatory przypisania.

Zmiany w klasie List

Dodajemy metodę `move()`. Porównując z wcześniej zaimplementowaną funckją `copy()`:

- `move()` przenosi wskaźniki na pierwszy i ostatni element listy. Oryginalny ciąg elementów nie zmienia się, ale zmienia się jego właściciel.
- `copy()` iteruje przez drugą listę i dodaje wszystkie wartości

```
void List::move(List&other){
    start=other.start;
    end=other.end;
    size = other.size;
    other.start=other.end = nullptr;
    other.size=0;
}

void List::copy(const List&other){
    start=end=0;
    for(ListElement*i=other.start;i!=0;i=i->next){
        pushBack(i->value);
    }
}
```

Zmiany w klasie List

Funkcję `move()` wykorzystujemy w implementacji przeciążonych wersji konstruktora kopiującego i operatora przypisania

```
List::List(List&&other){
    move(other);
}

List&List::operator=(List&&other){
    if(&other!=this){
        free();
        move(other);
    }
    return *this;
}
```

Mechanizm pomijania kopiowania

Pomijanie kopiowania (ang. copy elision) to kolejny mechanizm optymalizacji kodu.

Elizja to pomijanie przy wymawianiu głosek lub części wyrazów.

Wykorzystamy wcześniej wprowadzoną klasę drukującą adresy i konteksty tworzenia obiektów.

```
class A{
    string ctx;
public:
    A(const char*_ctx):ctx(_ctx){
        cout<<"Ctor ["<<ctx<<"] "<<hex<<this<<endl;
    }
…
};
```

Mechanizm pomijania kopiowania

```
A f_out(){
    A a("in_f_out");
    return a;
}

int main(){
    A a2 = f_out();
    cout<<"a2 in main():"<<&a2<<endl;
}
```

```
Ctor [in_f_out] 0xffffcc08
a2 in main():0xffffcc08
```

Standardowa implementacja instrukcji `A a2 = f_out();` przed C++11 polegała na:

- Skopiowaniu zawartości zwracanego obiektu (tu `a`) do nienazwanego obiektu tymczasowego (`tmp`)
- Wywołaniu destruktora zwracanego obiektu `a`
- Użyciu tmp jako argumentu konstruktora kopiującego tworzącego `a2`
- Wywołaniu destruktora `tmp`

Po elizji kopiowania obiekty `a` w `f_out()` i `a2` w `main()` mają te same adresy. A więc są to te same obiekty. Konstruktor jest uruchamiany tylko jeden raz i działa na pamięci wewnątrz ramki funkcji `main()`

Mechanizm pomijania kopiowania

```
int main(){
    A a2 = f_out();
    A a3("a3 in main");
    a3 =  f_out();
    cout<<"a2 in main():"<<&a2<<endl;

}
```

Mechanizm elizji działa dla konstruktorów kopiujących, a nie operatorów przypisania.

```
Ctor [in_f_out] 0xffffcc00
Ctor [a3 in main] 0xffffcbf8
Ctor [in_f_out] 0xffffcc08
move_assign
[in_f_out_assignment_of(0xffffcc08)]
0xffffcbf8
a2 in main():0xffffcc00
```

Jeżeli obiekt był wcześniej utworzony, miał zapewne jakąś zawartość, którą należy zwolnić stosując odpowiedni operator przypisania. W takim przypadku optymalizacja jest osiągana przez zastosowanie semantyki przenoszenia.