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: 03.04.2020

6. Przeciążanie funkcji i operatorów

Przeciążanie funkcji i operatorów

W języku C wymagane jest stosowanie unikalnych nazw funkcji.

Przykład

- Operacje `wash car` i `wash face` w języku C muszą być wyrażone jako `wash_car(struct car*)` oraz `wash_face(struct face*)`.
- W języku C++ możemy użyć tej samej nazwy w różnych kontekstach:

```
wash(struct car*)
wash(struct face*)
class Car{
   public: void wash();
};
```

Przestrzenie nazw

Dalszym udogodnieniem jest możliwość stosowanie przestrzeni nazw (ang. *namespace*). Umieszczając globalne funkcje w różnych przestrzeniach nazw mamy możliwość wielokrotnego definiowania funkcji o tych samych nazwach i argumentach.

```
// globalna funkcja identyfikowana jako ::f
void f(){}
namespace First{
        // globalna funkcja identyfikowana jako First::f
        void f(){}
}
namespace Second{
        // globalna funkcja identyfikowana jako Second::f
        void f(){}
}
```

W przestrzeniach nazw można także definiować klasy i deklarować zmienne. W pełni kwalifikowane nazwy są zawsze poprzedzane nazwą przestrzeni nazw i operatorem zasięgu :: .

Przeciążanie funkcji i operatorów

- Termin *przeciążanie funkcji* (*operatorów*) odnosi się do funkcji zdefiniowanych w tym samym zasięgu nazw (przestrzeni nazw lub w klasie).
- Interpretując wywołanie funkcji, kompilator wybierze wywołanie odpowiedniej implementacji na podstawie argumentów występujących w wywołaniu.

```
double max(double d1, double d2){
    return (d1>d2 ? d1:d2);
}
double max(int d1, int d2){
    return (d1>d2 ? d1:d2);
}

max(1.0,2.0) ; // wywoła wersję double
max(1,2) ; // wywoła wersję int
```

Rozróżnienie typów argumentów

- W trakcie wywołania przeładowanych funkcji kompilator wybiera wersję funkcji najlepiej pasującą do typu argumentów. Jeżeli odpowiednia funkcja zostanie odnaleziona, wówczas jest ona wołana.
- W przeciwnym przypadku kompilator będzie raportował niejednoznaczność traktowaną jako błąd.

```
// nieodróżnialne od double max(double d1, double d2)
double max(const double&d1, const double&d2){
    return (d1>d2 ? d1:d2);
}

// możliwe są dwie ścieżki automatycznej konwersji
max(1.0,2);
```

Dopasowania i konwersje

- **dokładne dopasowanie** argumentów wywołania do jednej z definicji
- następuje **trywialna konwersja**

`type-name`	`type-name&`
`type-name&`	`type-name`
`type-name[]`	`type-name*`
`type-name`	`const type-name`
`type-name*`	`const type-name*`

- następuje **konwersja całkowitoliczbowa** pomiędzy danymi typu `int, long, unsigned`
- istnieje **standardowa konwersja** pomiędzy argumentami

`void*`	`const void*`
`DerivedClass*`	`BaseClass*`
`DerivedClass&`	`BaseClass&`

- istnieją **zdefiniowane przez użytkownika** konwersje

`String`	`operator const char*()`
`const char*`	`String`

- w definicji funkcji pojawia się **elipsa** ...

Standardowe argumenty

Alternatywą do implementacji kilku przeciążonych wersji funkcji o podobnym zachowaniu jest zadeklarowanie jej standardowych argumentów.

```
int print(char *s ); // (1) Print a string.
// Print a double with default precision
int print(double dvalue); // (2)
// Print a double with a given precision.
int print(double dvalue, int prec);  // (3)
```

Funkcje (2)oraz (3) są bardzo do siebie podobne, funkcja (2) może być zaimplementowana jako

```
int print(double dvalue)
{
    return print(dvalue,DEFAULT_PRECISION);
}
```

Standardowe argumenty

Analogiczny efekt, jaki daje przeciążenie uzyskamy rezygnując z implementacji funkcji (2) oraz deklarując funkcję (3) jako:

```
int print(double dvalue, int prec = DEFAULT_PRECISION);
```

- Kompilator napotykając wywołanie: `print(4.5,3)` wywoła funkcję (3) z argumentem `prec = 3`.
- Napotykając wywołanie `print(5.1)` automatycznie wygeneruje wywołanie `print(5.1, DEFAULT_PRECISION)`.

Przeciążanie operatorów

Przeciążanie operatorów w C++ jest zabiegiem wyłącznie syntaktycznym. Wszystkie operatory są implementowane jako funkcje. Różnicą jest postać wywołania.

Zamiast pisać

```
x5 = plus(plus(plus(x1,x2),x3),x4) ;
```

możemy użyć zapisu

```
x5= x1+x2+x3+x4 ;
```

- Podobnie, jak w przypadku przeciążonych funkcji, kompilator automatycznie dobiera odpowiedni operator dokonując, jeżeli jest to wymagane, automatycznych konwersji.
- Nie można redefiniować trójargumenowego operatora warunkowego wyboru oraz kilu innych (. :: .*).

Składnia

- Operatory w C++ definiuje się z użyciem słowa kluczowego operator, po którym następuje nazwa operatora. Operatory mogą być składowymi klas lub mogą być zadeklarowane jako funkcje globalne.
- Liczba argumentów operatora uzależniona jest od jego typu oraz miejsca definicji.
- W C/C++ występują operatory:
  - Unarne (jednoargumentowe)
  - Binarne (dwuargumentowe)
  - Jeden operator terenarny (trójargumentowy) ?

```
z = x<0 ? 0 : x;
```

	Unarny	**Binarny**
Globalny	1 argument	2 argumenty
Lokalny (metoda klasy)	0 argumentów	1 argument

Składnia

- Składnia wywołania przeciążonych operatorów jest zgodna ze składnią operatorów C/C++ dla wbudowanych typów.
- Operatory nie mogą mieć standardowych argumentów
- Wszystkie przeciążone operatory (poza operatorem przypisania `operator=`) są dziedziczone.
- Podczas wywołania operatorów pierwszym argumentem musi być zawsze typ (referencja typu), dla której operator został zdefiniowany. **Kompilator nigdy nie stosuje konwersji dla pierwszego argumentu**.

Przykład

```
class String
{
public:
    char buf[256];
    String(const char*txt=""){strcpy(buf,txt);}
    operator const char*()const {return buf;}
    String&operator=(const String&s)   {
        strcpy(buf,s.buf);
        return *this;
    }
    char&operator[](int idx){
        if(idx<0 || idx>255)return buf[0];//throw 0;
        return buf[idx];
    }
};
```

Przykład

```
String&operator+=(String&s,const char*txt){
    strcat(s.buf,txt);
    return s;
}

const String operator+(const String&s,const char*txt) {
    String r(s.buf);
    strcat(r.buf,txt);
    return r;
}

bool operator==(const String&s1,const char*txt)
{
    return !strcmp(s1.buf,txt);
}
```

Przykład

```
int main(){
    String a("Ala ma");
    a+=" ";
    String b("kota");
    a+=b;  // automatycznie wywoła
    //String::operator const char*()
    if(a=="Ala ma kota"){…}
    b = b + " i psa";
    for(int i=0;i<strlen(b);i++)putchar(b[i]);
}
```

Operatory inkrementacji i dekrementacji

- Unarne (jednoargumentowe) operatory inkrementacji i dekrementacji występują w dwóch odmianach: prefiksowej i postfiksowej;
- Typowa implementacja jest następująca:

```
class Int{
    int value;
public:
    Int&operator++(){
        value++;return *this;
    }
    Int operator++(int){
        Int temp = *this;
        ++*this;
        return temp;
    }
};
```

Operator przypisania

- Operator przypisania operator= musi być zadeklarowany jako metoda klasy. Zazwyczaj deklaracja ma postać:

```
X&operator=(const X&)
```

- Operator ten nie jest dziedziczony, ponieważ musi skopiować *wszystkie* pola klasy.
- Dla wielu klas kompilator jest w stanie wygenerować automatyczny operator przypisania, który wywoła operatory przypisania poszczególnych atrybutów.

Przykład

```
class TwoStrings
{
public:
    String s1;
    String s2;
};

TwoStrings a,b;
a = b;
// automatycznie wywoła: //
// a.s1= b.s1;
// a.s2=b.s2;
```

Standardowa implementacja operatora przypisania dostarczona przez kompilator wywołuje operator przypisania dla kolejnych pól obiektu.

Jeżeli te pola nie są wskaźnikami, zazwyczaj implementacja własnego operatora przypisania nie jest konieczna.

Operator przypisania powinien być definiowany dla klas **alokujących pamięć**. Zazwyczaj także definiujemy wtedy konstruktor kopiujący.

Modelowy przykład klasy alokującej pamięć

```
class Array
{
    double *data;
    int size;
public:
    Array(int s=0):size(s),data(0){
        if(size>0)data = new double[size];
    }
    Array(const Array&other){
        copy(other);
    }
    ~Array(){free();}
    Array&operator=(const Array&other){
        if(&other != this){
            free();
            copy(other);
        }
        return *this;
    }
protected:
    void free();
    void copy(const Array&other);
};
```

Modelowy przykład klasy alokującej pamięć

```
void Array::free(){
    if(data)delete []data;
    data =0;
    size=0;
}
void Array::copy(const Array&other){
    size = other.size;data=0;
    if(size>0)data = new double[size];
    for(int i=0;i<size;i++)data[i]=other.data[i];
}
```

`if(&other != this)` – zabezpiecza przed zwolnieniem pamięci obiektu, dla bezpośredniego lub pośredniego wywołania przypisania `x = x`.

Operatory ekstrakcji i wstawiania

Zazwyczaj definiuje się w celu odczytu lub zapisu zawartości obiektów z/do strumieni.

```
Vector v;
cin>>v;
cout<<v;
```

Ponieważ pierwszym argumentem jest strumień, operatory mogłyby być zaimplementowane:

- jako **metoda strumienia** z jednym parametrem typu `Vector&`
- jako **globalna funkcja z dwoma parametrami**

Ponieważ nie modyfikujemy klas bibliotecznych, **zawsze są implementowane jako funkcje globalne**.

Przykład

```
#include <iostream>
using namespace std;

class Array
{
    friend ostream&operator<<(ostream&os,const Array&v);
    friend istream&operator>>(istream&is, Array&v);
    double *data;
    int size;
public:
    // pozostałe deklaracje
};
```

Deklarujemy operatory jako funkcje zaprzyjaźnione (`friend`). W ten sposób będą miały dostęp do pól prywatnych.

Przykład

```
ostream&operator<<(ostream&os,const Array&v){
    os<<v.size<<" ";
    for(int i=0;i<v.size;i++){
        os<<v.data[i]<<" ";
    }
    return os;
}

istream&operator>>(istream&is, Array&v){
    if(!is)return is;
    v.free();
    is>>v.size;
    v.data = new double[v.size];
    for(int i=0;i<v.size;i++){
        is>>v.data[i];
    }
    return is;
}
```

Zapisujemy rozmiar,

a następnie wszystkie dane

1. Czyścimy zawartość,
2. Wczytujemy rozmiar
3. Alokujemy pamięć
4. Wczytujemy dane do tablicy

Operator wywołania funkcji

Tą nazwą określa się operator ( ) . Operator ten jest operatorem dwuargumentowym i ma postać

```
expression (expression-list)
```

`expression` – jest zazwyczaj nazwą funkcji,

`expression-list` – listą argumentów.

```
class Matrix
{
    double e[100][100];
public:
    double&operator()(int row,int col)
    {
        return e[row][col];
    }
};
```

```
int main(){
    Matrix m;
    m(2,3)=7.5;
    printf("%f",m(2,3));
}
```

Obiekty funkcyjne

```
class Function{
public:
    virtual double operator()(double x)const{return 0;}
};

class SinFunction:public Function{
public:
    double operator()(double x)const{return sin(x);}
};

void call(const Function&f){
    cout<<f(M_PI/6)<<endl;
}

int main(){
    SinFunction sin;
    call(sin);
}
```

- Zdefiniowano klasę `Function` z **wirtualnym** operatorem wywołania funkcji. Jest to tak zwany obiekt funkcyjny (ang. *function object*)
- Klasa `SinFunction` dziedziczy po `Function`.

Oczywiście,wynik to: $\sin(\pi) = 0.5$

Obiekty funkcyjne

```
class PolyFunction: public Function{
    std::vector<double> a;
public:
    PolyFunction(double tab[],int n);
    PolyFunction(int n,...);
    double operator()(double x)const;
};


PolyFunction::PolyFunction(double tab[],int n){
    for ( int i = 0; i < n; i++ )
        a.push_back(tab[i]);
}
```

Funkcja z zmienną liczbą argumentów (ang. *variadic*)

- Obiekt funkcyjny może też przechowywać dane, np. `PolyFunction` to wielomian.
- W atrybucie a przechowywane są współczynniki wielomianu. `std::vector` to biblioteczna klasa – dynamicznie przyrastająca tablica, podobna do `Array`. Metoda `push_back` dodaje watrtość na końcu tablicy.

Funkcja typu variadic

```
#include <stdarg.h>

PolyFunction::PolyFunction(int n,...){
    va_list arguments;
    va_start ( arguments, n );
    for ( int i = 0; i < n; i++ )
        a.push_back(va_arg ( arguments, double ));
    va_end ( arguments );
}
```

- Deklaracja funkcja ze zmianną (nieokreśloną) liczbą argumentów wymaga podania co najmniej jednego parametru, po którym występują trzy kropki.
- Do odczytu rzeczywistych argumentów wywołania stosuje się makra prerprocesora zdefiniowane w pliku `stdarg.h`
- Funkcja musi wiedzieć jakiego typu są argumenty -- odczytuje je za pomoca makra `va_arg(lista,typ_danych)`

Obiekty funkcyjne

```
double PolyFunction::operator()(double x)const{
    double r=0;
    double pow=1;
    for(int i=a.size()-1;i>=0;i--){
        r+=a[i]*pow;
        pow*=x;
    }
    return r;
}


int main(){
    double t[]={1,2,3,4,5};
    PolyFunction poly1(t,5);
    cout<<poly1(0)<<endl;
    PolyFunction poly2(4,1.0,-2.0,3.0,4.0);
    cout<<poly2(1);
}
```

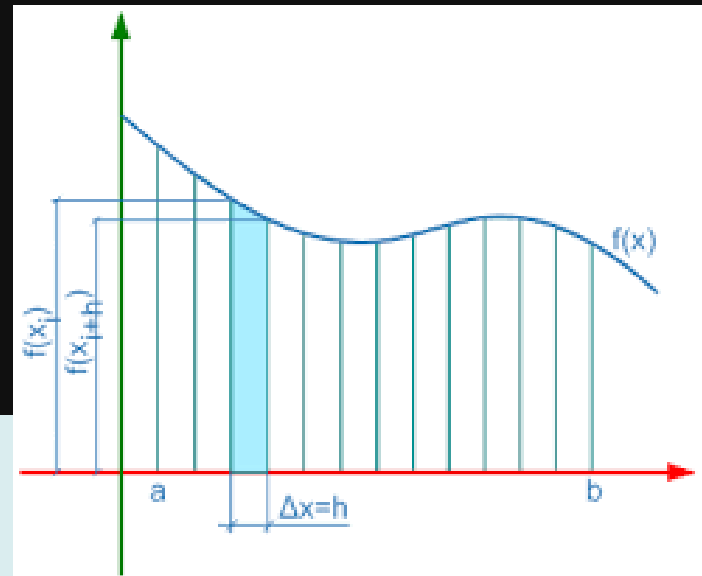
`operator()` – do zwracanego wyniku `r` dodawane są kolejne potęgi argumentu x pomnożone przez współczynniki przechowywane w wektorze a.

Wynik: 5 i 6

W wywołaniu funkcji variadic typy muszą się zgadzać. Trochę niewygodne?

Obiekty funkcyjne - zastosowanie

```
double integrate(const Function&f,double a, double b,
                                         int steps=1000){
    double delta = (b-a)/steps;
    double sum=0;
    double fs = f(a);
    for(int i=1;i<=steps;i++){
        double fe=f(a+i*delta);
        sum+=(fs+fe)/2*delta;
        fs=fe;
    }
    return sum;
}
```



Obiekt funkcyjny przekazywany jest do funkcji ogólnego zastosowania służącej do numerycznego obliczania całki metodą trapezów.

[https://www.obliczeniowo.com.pl/704]

Obiekty funkcyjne - zastosowanie

```
int main(){
    double t[]={1,2,3,4,5};
    PolyFunction poly1(t,5);
    cout<<integrate(poly1,0,10)<<endl;
    cout<<integrate(PolyFunction(4,1.0,-2.0,3.0,4.0),0,10)<<endl;
    cout<<integrate(SinFunction(),0,M_PI)<<endl;
}
```

Wynik:
26250
2023.34
2

[https://live.sympy.org/]

```
>>> integrate(1*x**4+2*x**3+3*x**2+4*x+5,(x,0,10))
```

26250

```
>>> integrate(1.0*x**3-2.0*x**2+3.0*x+4.0,(x,0,10))
```

2023.33333333333

```
>>> integrate(sin(x),(x,0,pi))
```

2

Operator dostępu do składowych

Operator ten jest definiowany jako

```
class-type*operator->()
```

Musi być on metodą pewnej klasy, która pełni rolę „sprytnego” lub „inteligentnego” wskaźnika (ang. *smart pointer*) pozwalającego na dostęp do pól i metod obiektu `class-type`.

Podobne funkcje może pełnić operator dereferencji zdefiniowany jako:

```
class-type&operator*()
```

Operator powinien zwrócić referencję do obiektu wskazywanego przez obiekt `SmartPointer`.

Zazwyczaj obiekt typu `SmartPointer` realizuje dodatkowe zabezpieczenia lub zlicza referencje do obiektu, co pozwala na automatyczne usuwanie nieużywanych obiektów.

SmartPointer - przykład klasyczny

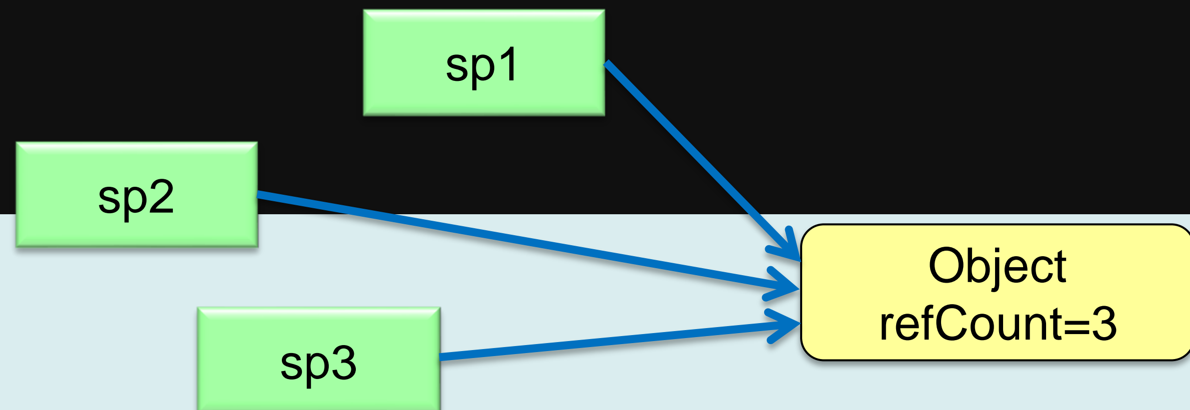
```
class Object
{
    int refCount;
    friend class SmartPointer;
public:
    Object(){refCount=0;}
    void dump()const{
        printf("Object.refcount=%d\n",refCount);
    }
};
```

- Obiekty typu `SmartPointer` będą zachowywały się, jak wskaźniki do klasy `Object` (lub klas potomnych).
- W klasie `Object` zaimplementowany jest licznik referencji `refCount`. Kiedy kolejny wskaźnik `SmartPointer` będzie przekierowany na obiekt, jego liczba referencji będzie inkrementowana
- Kiedy `SmartPointer` przestanie wskazywać obiekt – liczba referencji obiektu będzie zmniejszana.
- Kiedy `refCount` osiągnie 0 – obiekt zostanie automatycznie usunięty.

SmartPointer - przykład klasyczny

```
class SmartPointer
{
    Object*obj;
    void bind(Object*_obj){
        obj=_obj;
        if(obj)obj->refCount++;
    }
    void release(){
        if(obj){
            obj->refCount--;
            if(obj->refCount==0)delete obj;
        }
    }
public:
//...
```

- `bind()` – wiąże `SmartPointer` z obiektem
- `release()` – usuwa powiązanie, zmniejsza `refCount` i opcjonalnie usuwa obiekt



SmartPointer - przykład klasyczny

```
//...
public:
    // konstruktor
    SmartPointer(Object*_obj=0){
        bind(_obj);
    }
    // konstruktor kopiujący
    SmartPointer(const SmartPointer&p){
        bind(p.obj);
    }
    // Destruktor
    ~SmartPointer(){release();}
    // Operator przypisania dla wskaźnika
    SmartPointer&operator=(Object*_obj){
        if(obj!=_obj){ release(); bind(_obj);}
        return *this;
    }
    // Operator przypisania dla "opakowanego" wskaźnika
    SmartPointer&operator=(const SmartPointer&p){
        if(p.obj!=obj){ release(); bind(p.obj); }
        return *this;
    }
//...
```

Kilka typowych funkcji: konstruktory, destruktory, operatory przypisania…

- Konstruktory wołają `bind()`
- Destruktor: `release()`
- Operatory przypisania `release() i bind()`; są też zabezpieczone przed przypadkowym usunięciem obiektu.

SmartPointer - przykład klasyczny

```
class SmartPointer{
// ...
public:
// ...
    Object*operator->(){return obj;}

    class NullPointerException{};
    Object&operator*(){
        if(!obj)throw NullPointerException();
        return *obj;
    }
};
```

- Przeciążone operatory `->` oraz `*` zwracają po prostu wskaźnik `obj` lub `*obj`.
- Jeżeli wskaźnik obj ma wartość 0 (NULL, nullptr) żadna sensowna wartość nie może zostać zwrócona. Generowany jest wyjątek…

SmartPointer - przykład klasyczny

```
int main()
{
    SmartPointer sp1=new Object();
    sp1->dump(); // Object.refcount=1
    {
        SmartPointer sp2=sp1;
        sp2->dump(); // Object.refcount=2
    }
    (*sp1).dump(); // Object.refcount=1
} // obiekt jest usuwany
```

Przez wiele lat inteligentne wskaźniki były używane głownie do implementacji iteratorów.

W C++11 pojawiły się jednak biblioteczne implementacje: `shared_pointer`, `unique_pointer` i `weak_pointer`.

Wynik:
`Object.refcount=1`
`Object.refcount=2`
`Object.refcount=1`
Przy wyjściu z funkcji main() destruktor sp1 spowoduje usunięcie obiektu

SmartPointer - przykład klasyczny

```
int main()
{
    SmartPointer sp1=new Object();
    sp1->dump(); // Object.refcount=1
    {
        SmartPointer sp2=sp1;
        sp2->dump(); // Object.refcount=2
    }
    (*sp1).dump(); // Object.refcount=1
} // obiekt jest usuwany
```

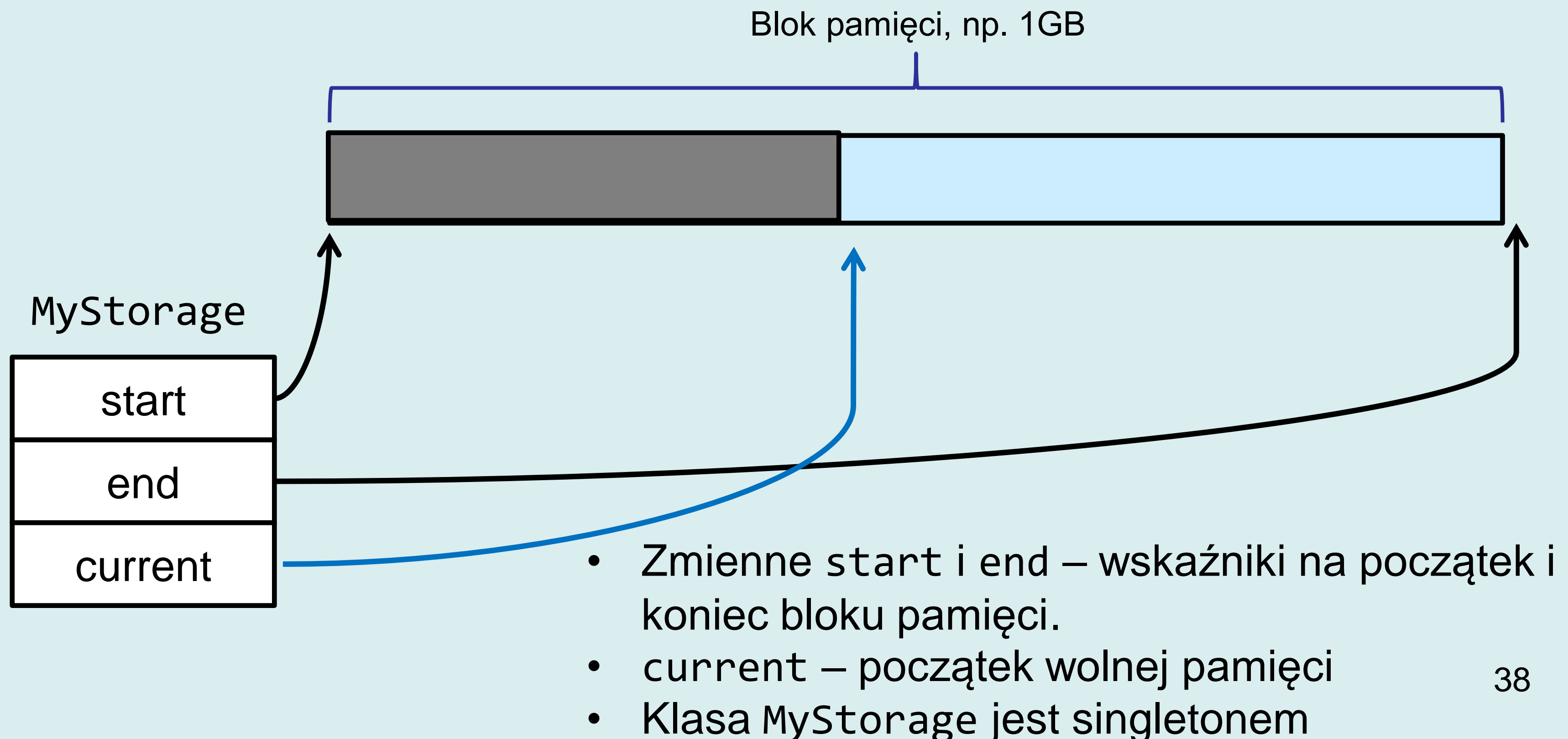
Przez wiele lat inteligentne wskaźniki były używane głownie do implementacji iteratorów.

W C++11 pojawiły się jednak biblioteczne implementacje: `shared_pointer`, `unique_pointer` i `weak_pointer`.

Wynik:
```
Object.refcount=1
Object.refcount=2
Object.refcount=1
```
Przy wyjściu z funkcji main() destruktor sp1 spowoduje usunięcie obiektu

Operatory new i delete

Możliwe jest przeciążenie globalnych i lokalnych operatorów `new` i `delete`, np.: aby przyspieszyć alokację pamięci lub pominąć w kodzie jej zwalnianie (cały blok zostanie zwolniony przy końcu programu).



- Zmienne `start` i `end` – wskaźniki na początek i koniec bloku pamięci.
- `current` – początek wolnej pamięci
- Klasa `MyStorage` jest singletonem

Klasa do zarządzania pamięcią

```
class MyStorage{
    static void*start;
    static void*end;
    static void*current;
    MyStorage(){initAllocator();}
    ~MyStorage(){freeAllocator();}
    static MyStorage _myStorage;
public:
    static void initAllocator();
    static void freeAllocator();
    static void*alloc(size_t size);
    static void free(void*){}
    static void reset(){current=start;}
};

void *MyStorage::start=0;
void *MyStorage::end=0;
void *MyStorage::current=0;
MyStorage MyStorage::_myStorage;
```

Klasa do zarządzania pamięcią

```
void MyStorage::initAllocator(){
    if(start)return;
    size_t size=1024;//1024*1024*1024; // 1GB
    start=malloc(size);
    current=start;
    if(start){end = (char*)start+size;}
    printf("start:%p\n",start);
    printf("end:%p\n",end);
    printf("end-start:%d\n",(char*)end-(char*)start);
}

void MyStorage::freeAllocator(){
    if(start)::free(start);
    start=0;
}
```

`initAllocator()` – przydziela pamięć dla dużego bloku, np. 1GB
`freeAllocator()` – zwalania cały blok (wołane w destruktorze)

Klasa do zarządzania pamięcią

```
void*MyStorage::alloc(size_t size)
{
    if(start==0)throw std::bad_alloc();
    printf("\n%p --> ",current);
    printf("%u ",size);

    void*old=current;
    size_t wordsize = sizeof(void*);
    current = (char*)current +
            wordsize*(size/wordsize + (size%wordsize>0?1:0));
    if(current>end)throw std::bad_alloc();
    printf(" (%p) ",current);

    return old;
}
```

`throw std::bad_alloc();` Wyjątek informujący o braku pamięci

- `alloc()` – przydziela pamięć o rozmiarze size wewnątrz dużego bloku
- Wielkość bloku jest zaokrąglana w górę do wielokrotności `wordsize` – rozmiaru słowa maszynowego na danej platformie (8B na platformie 64-bitowej)

Przeciążone operatory

```
void *operator new(size_t size){
    return MyStorage::alloc(size);
}

void *operator new[](size_t size){
    return MyStorage::alloc(size);
}

void operator delete(void*block){
    return MyStorage::free(block); // nic nie robi
}

void operator delete[](void*block){
    return MyStorage::free(block); // nic nie robi
}
```

Test

```
class A{
public:
    A(){printf(" this:%p",this);}
};

int main(int argc, char *argv[])
{
    for(int i=0;;i++){
        try{
            char*t=new char[256];
            strcpy(t,"Ala ma kota");
            delete []t;
            new A();
            A*a=new A[3];
        }
        catch(std::bad_alloc&){
            cout<<"No storage at "<<i<<endl;
            break;
        }
    }
}
```

Wyniki – rozmiar bloku 1024

```
for(int i=0;;i++){
    char*t=new char[256];
    strcpy(t,"Ala ma kota");
    delete []t;
    new A();
    A*a=new A[3];
}
```

```
start:0x600012850
end:0x600012c50
end-start:1024

0x600012850 --> 256  (0x600012950)
0x600012950 --> 1  (0x600012958)  this:0x600012950
0x600012958 --> 3  (0x600012960)  this:0x600012958 this:0x600012959 this:0x60001295a
0x600012960 --> 256  (0x600012a60)
0x600012a60 --> 1  (0x600012a68)  this:0x600012a60
0x600012a68 --> 3  (0x600012a70)  this:0x600012a68 this:0x600012a69 this:0x600012a6a
0x600012a70 --> 256  (0x600012b70)
0x600012b70 --> 1  (0x600012b78)  this:0x600012b70
0x600012b78 --> 3  (0x600012b80)  this:0x600012b78 this:0x600012b79 this:0x600012b7a
0x600012b80 --> 256 No storage at 3
```

Test wydajności

```
double  allocation_time(){
    clock_t t0 = clock();
    for(int i=0;i<4*1024*1024;i++) {
            char *t = new char[256];
            delete[]t;
    }
    clock_t t1 = clock();
    double texec =  (t1 - t0) * (1.0 / CLOCKS_PER_SEC);
    MyStorage::reset();
    return texec;
}
double mean(double*t,int n );
double stddev(double*t,int n );

int main(int argc, char *argv[]){
    double t[10];
    for(int i=0;i<10;i++)
        t[i]=allocation_time();
    printf("mean=%f std=%f\n",mean(t,10),stddev(t,10));
}
```

Własny alokator (rozmiar bloku 1GB):
`mean=0.101700 std=0.008341`

Standardowy alokator:
`mean=0.606200 std=0.009739`