

# Dzisiejszy wykład

- # Wskaźniki
- # Klasa listy jednokierunkowej
- # Przekazywanie parametrów do funkcji
- # Płytkie i głębokie kopiowanie
- # Konstruktor kopiujący
- # Operator przypisania
- # Przeciążenie operatorów

# Wskaźniki

## # Cztery atrybuty zmiennej

- nazwa
- typ
- wartość
- lokacja (adres)

```
int x = 5;
```

## # Wskaźnik jest typem wartości

- przechowywany w zmiennej
- jest to pewna liczba

## # Operator \* oznacza:

- pobierz wartość przechowywaną w zmiennej i użyj jej jako adresu innej zmiennej

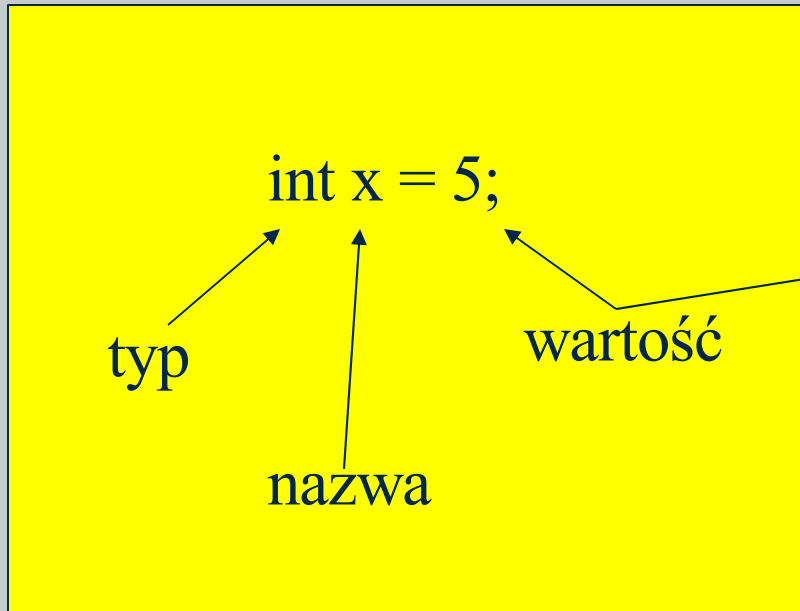
## # Operator & oznacza:

- pobierz adres zmiennej (NIE wartość zmiennej)

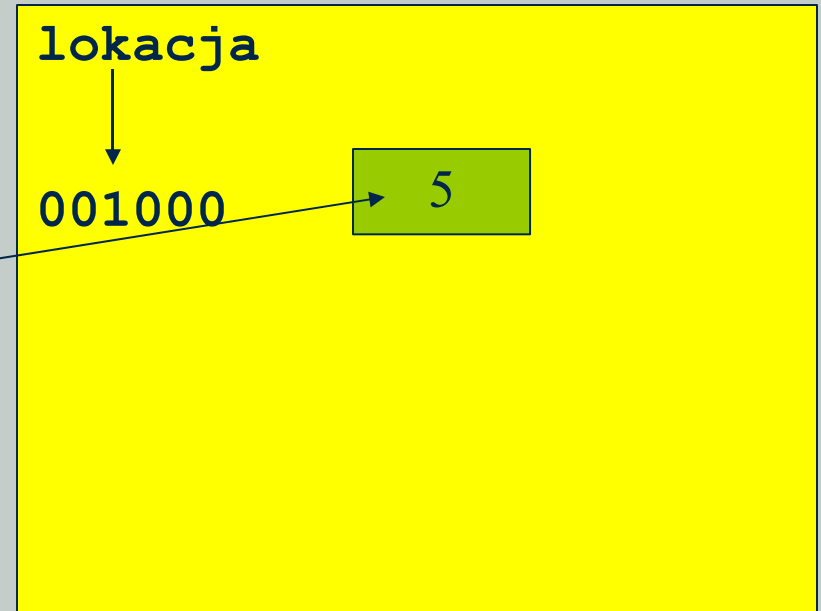
# Wskaźniki

## # Zmienna

- nazwa, typ, wartość, lokacja (adres)



W programie



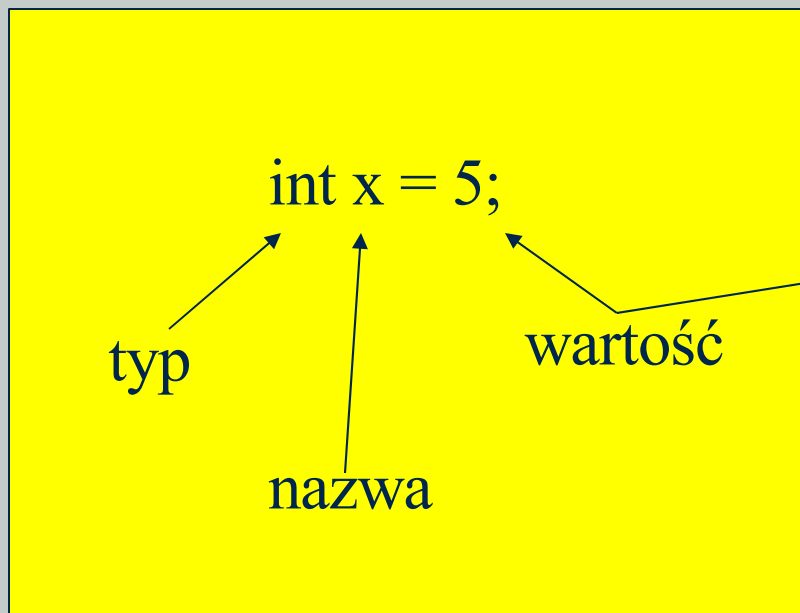
W pamięci

- # Która zmienna pod jakim adresem? Ile pamięci zajmuje? Kto o tym decyduje?

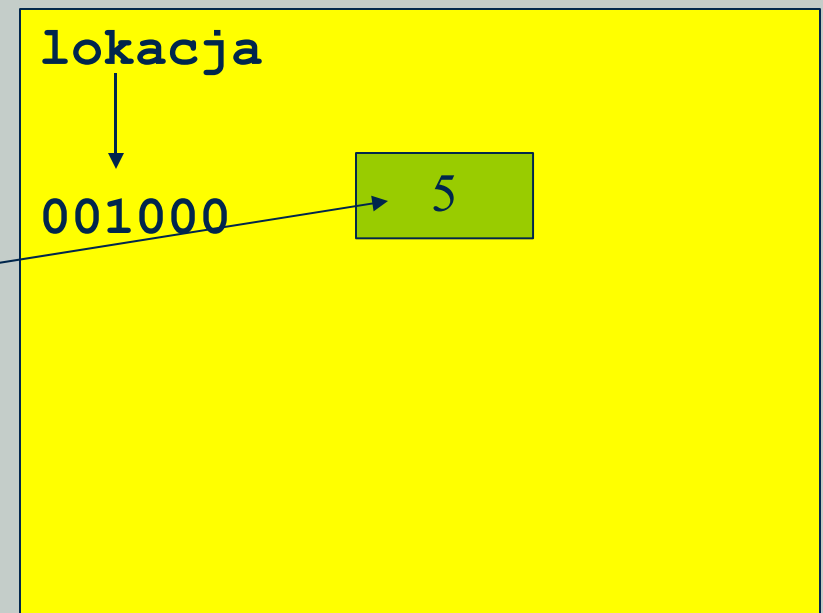
# Wskaźniki

# Jaka jest wartość poniższych wyrażeń? Czy są one poprawne?

- `x`
- `&x`
- `*x`



W programie



W pamięci

# Wskaźniki

# Jaka jest wartość poniższych wyrażeń? Czy są one poprawne?

- `x, &x, *x`
- `p, &p, *p`
- `q, &q, *q`
- `ip, &ip, *ip`

```
int x=5;
char *p="hello";
char *q;
int *ip;
ip=&x;
```

W programie

lokacja	wartość	nazwa
---------	---------	-------

001000	5	int x
001004	3000	char*p
001008	?	char*q
001012	?	int* ip
...	...	
003000	hello\0	

W pamięci

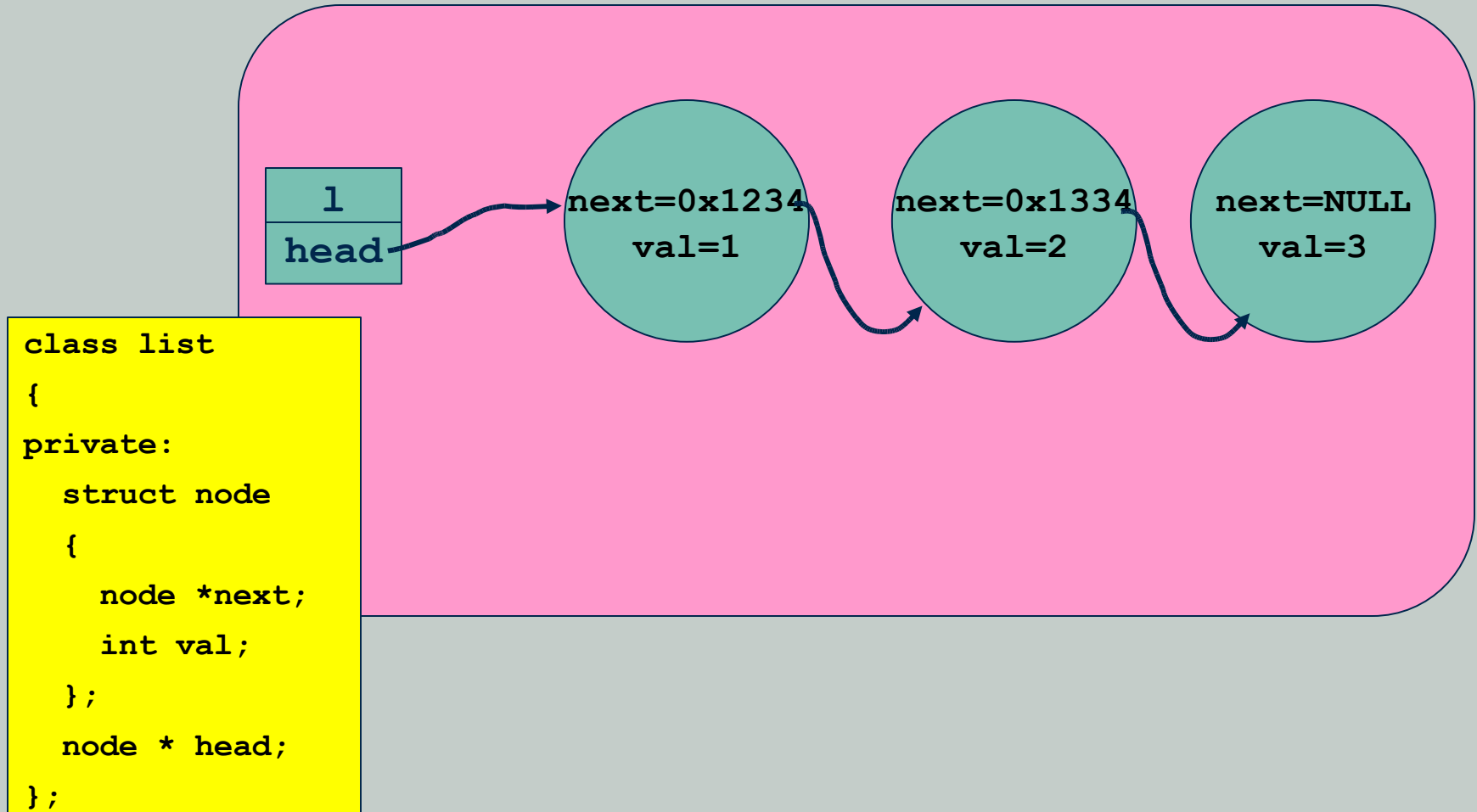
# Wskaźniki do funkcji

```
int* f1(int*, const int*);  
int* (*fp1)(int*, const int*);  
int* (*f2(int))(int*, const int*);  
int* ((*fp2)(int))(int*, const int*);  
  
fp1=f1;  
fp1=&f1;  
fp1=&fp1; /* zle */  
fp2=f2;  
fp2=&f2;
```

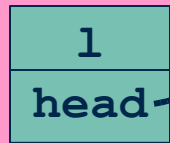
```
int a,b,*c;  
  
c=f1 (&a , &b) ;  
c=fp1 (&a , &b) ;  
c=(*fp1) (&a , &b) ;  
c=*fp1 (&a , &b) ; /* zle */  
c=(f2 (3) ) (&a , &b) ;  
c=(*f2 (3) ) (&a , &b) ;  
c=(fp2 (3) ) (&a , &b) ;  
c=(*fp2 (3) ) (&a , &b) ;  
c=(* (*fp2) (3) ) (&a , &b) ;
```

# Klasa List

## # Lista jednokierunkowa z dowiązaniem



# Klasa List - konstruktor i destruktor



next=0x1234  
val=1

next=0x1334  
val=2

next=NULL  
val=3

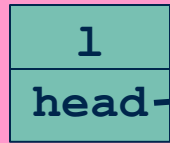
```
class list
{
private:
    struct node
    {
        node *next;
        int val;
    };
    node * head;
public:
    list ();
    ~list ();
    void insert (int a);
}
```

```
list::list()
{
    head = NULL;
}
```

```
list::~~list()
{
    while(head)
    {
        node* t=head->next;
        delete head;
        head=t;
    };
}
```



# Klasa List - destruktor



next=0x1334  
val=2

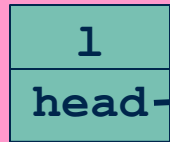
next=NULL  
val=3

```
class list
{
private:
    struct node
    {
        node *next;
        int val;
    };
    node * head;
public:
    list ();
    ~list ();
    void insert (int a);
}
```

```
list::list()
{
    head = NULL;
}
```

```
list::~~list()
{
    while(head)
    {
        node* t=head->next;
        delete head;
        head=t;
    }
}
```

# Klasa List - destruktor



```
class list
{
private:
    struct node
    {
        node *next;
        int val;
    };
    node * head;
public:
    list ();
    ~list ();
    void insert (int a);
}
```

```
list::list()
{
    head = NULL;
}
```

```
list::~~list()
{
    while(head)
    {
        node* t=head->next;
        delete head;
        head=t;
    };
}
```

# Klasa List - destruktor

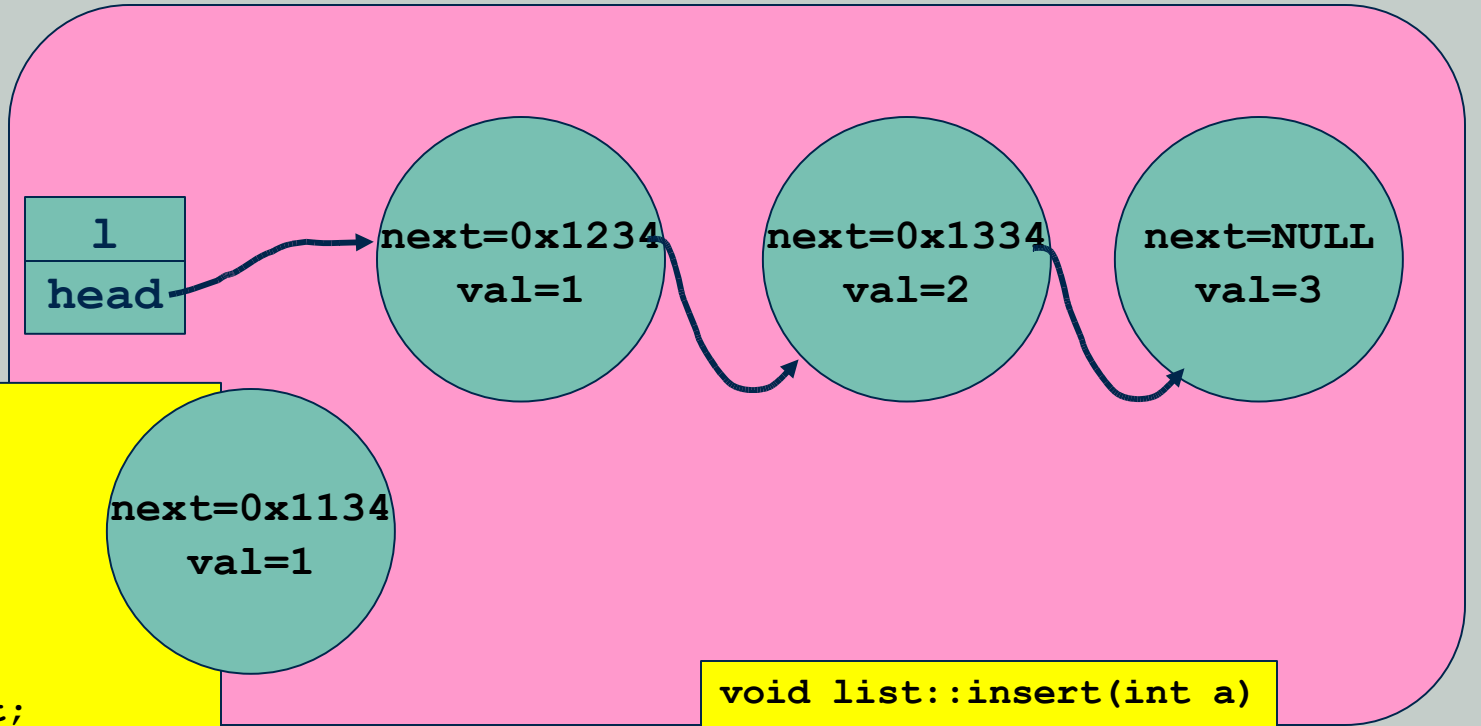
1  
head=NULL

```
class list
{
private:
    struct node
    {
        node *next;
        int val;
    };
    node * head;
public:
    list ();
    ~list ();
    void insert (int a);
}
```

```
list::list()
{
    head = NULL;
}
```

```
list::~~list()
{
    while(head)
    {
        node* t=head->next;
        delete head;
        head=t;
    };
}
```

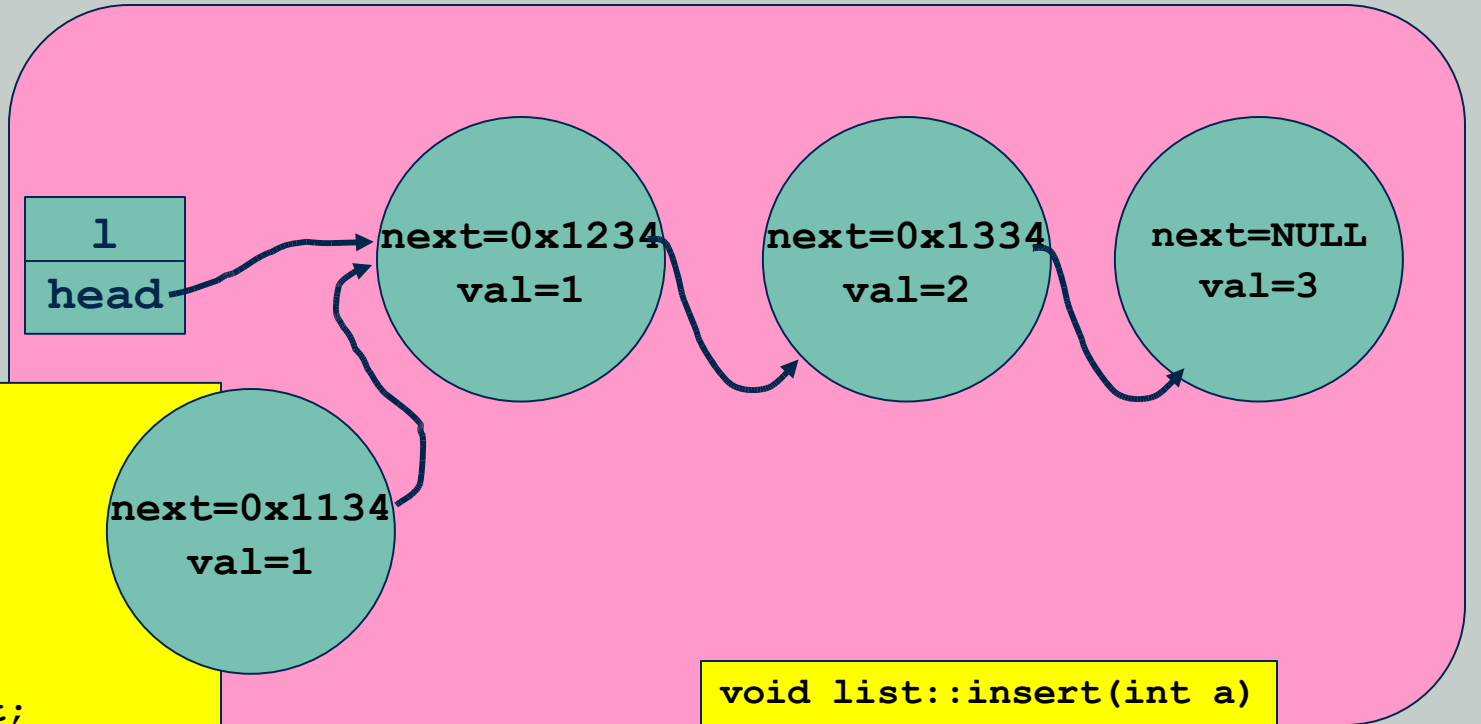
# Klasa List - insert



```
class list
{
private:
    struct node
    {
        node *next;
        int val;
    };
    node * head;
public:
    list ();
    ~list ();
    void insert (int a);
}
```

```
void list::insert(int a)
{
    node* nowy=new node;
    nowy->next=head;
    head = nowy;
    head->val = a;
}
```

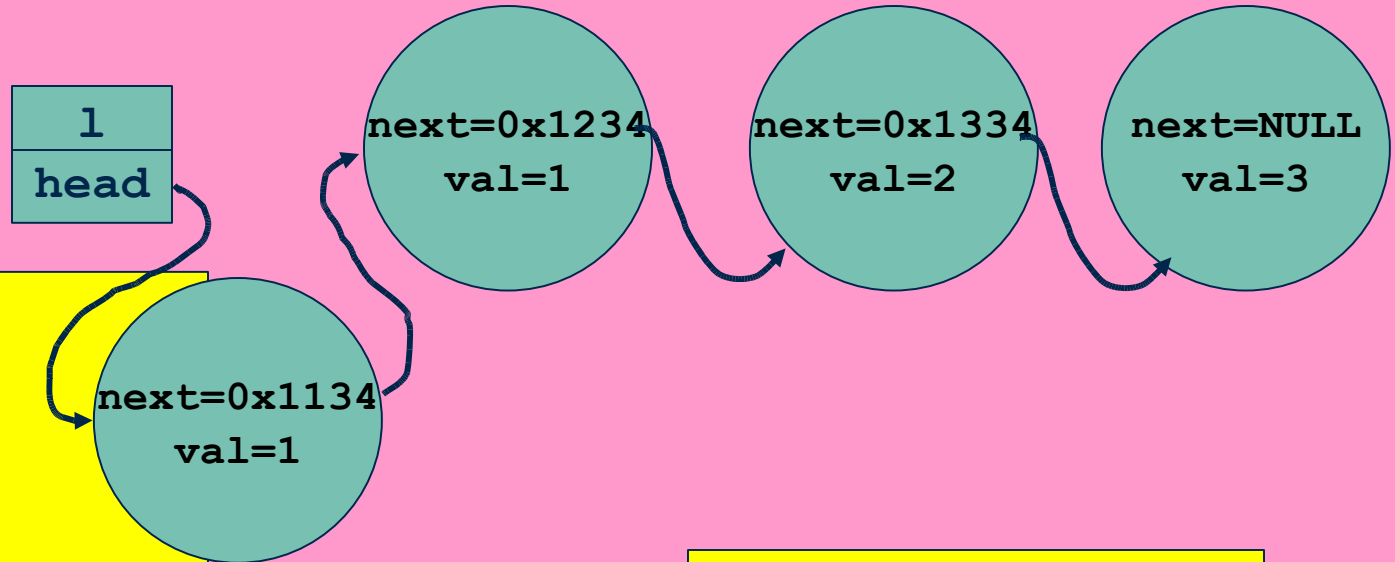
# Klasa List - insert



```
class list
{
private:
    struct node
    {
        node *next;
        int val;
    };
    node * head;
public:
    list ();
    ~list ();
    void insert (int a);
}
```

```
void list::insert(int a)
{
    node* nowy=new node;
    nowy->next=head;
    head = nowy;
    head->val = a;
}
```

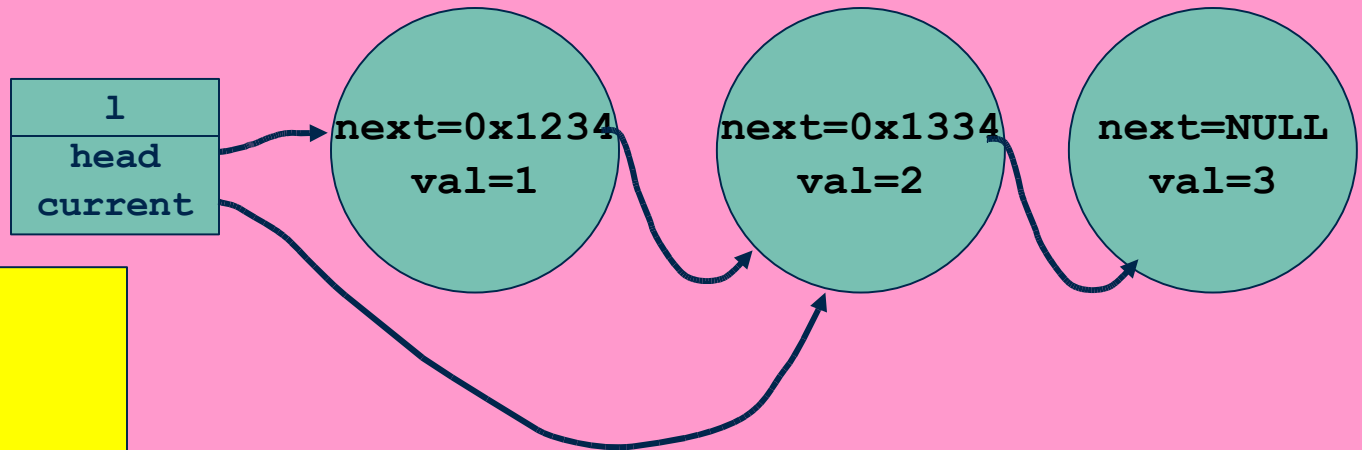
# Klasa List - insert



```
class list
{
private:
    struct node
    {
        node *next;
        int val;
    };
    node * head;
public:
    list ();
    ~list ();
    void insert (int a);
}
```

```
void list::insert(int a)
{
    node* nowy=new node;
    nowy->next=head;
    head = nowy;
    head->val = a;
}
```

# Klasa List - iterator



```
class list
{
private:
...
    node * head;
    node *current;
public:
...
    void goToHead ();
    int getCurrentData
();
    void advance ();
    bool moreData ();
};
```

```
#include <iostream>
using namespace std;
#include "list.h"
int main()
{
    list l;
    l.insert(3);
    l.insert(2);
    l.insert(1);
    l.goToHead();
```

```
while(l.moreData())
{
    int val;
    val=l.getCurrentData();
    cout << val << " ";
    l.advance();
}
cout << endl;
};
```

# Przekazywanie argumentów do funkcji

## # Przekazywanie przez wartość

- parametry formalne są kopią parametrów aktualnych

## # Przekazywanie przez referencję

- parametry formalne są referencją do parametrów aktualnych, tj. wszystkie operacje na parametrach formalnych odnoszą się do parametrów aktualnych

## # C i C++ domyślnie przekazują parametry przez wartość

```
void d1(int x)
{ x = 10; }
void d2(int *p)
{ (*p) = 10;}
void d3(int *p)
{ p = new int(4);}
```

```
int main() {
    int y = 2;
    d1(y); cout << y;
    d2(&y); cout << y;
    d3(&y); cout << y;
}
```



# Przekazywanie argumentów do funkcji

## Przez wartość

- parametry formalne są kopią parametrów aktualnych

## Przez referencję

- parametry formalne są referencją do parametrów aktualnych, tj. wszystkie operacje na parametrach formalnych odnoszą się do parametrów aktualnych

## Przez stałą referencję

- do funkcji przekazywana jest referencja do parametru w celu uniknięcia kosztów kopiowania, ale wartość nie może być w funkcji modyfikowana (co jest sprawdzane przez kompilator)

```
void f1(int x) { x = x + 1; }
void f2(int& x) { x = x + 1; }
void f3(const int& x) { x = x + 1; }
void f4(int *x) { *x = *x + 1; }
int main() {
    int y = 5;
    f1(y);
    f2(y);
    f3(y);
    f4(&y);
}
```

- Która metoda w którym przykładzie?
- Ile wynosi y po każdym wywołaniu?
- Co jest przekazywane do f4? Czy to wartość, czy referencja?
- Czy można przekazać wskaźnik przez referencję?

# Przekazywanie argumentów do funkcji

- ✚ Przekazywanie obiektów do funkcji nie różni się koncepcyjnie od przekazywania typów prostych
  - klasy umożliwiają modyfikację zachowania w tym przypadku
- ✚ Trzy metody: przez wartość, przez referencję, przez stałą referencję
- ✚ Przez wartość
  - parametr formalny jest kopią parametru aktualnego. Użyty konstruktor kopiujący.
- ✚ Przez referencję
  - parametry formalne są referencją do parametrów aktualnych, tj. wszystkie operacje na parametrach formalnych odnoszą się do parametrów aktualnych
- ✚ Przez stałą referencję
  - do funkcji przekazywana jest stała referencja do argumentu. Tylko metody ze specyfikatorem *const* mogą być wywoływane wewnątrz metody na rzecz argumentu.

# Przekazywanie obiektów jako parametrów

- ⚠ Kiedy obiekt jest użyty jako parametr wywołania funkcji, rozróżnienie między kopiowaniem płytkim a głębokim może spowodować tajemnicze problemy

```
void
PrintList (list & toPrint, ostream & Out)
{
    int nextValue;
    Out << "Printing list contents: " << endl;
    toPrint.goToHead ();
    if (!toPrint.moreData ())
        {
            Out << "List is empty" << endl;
            return;
        }
    while (toPrint.moreData ())
        {
            nextValue = toPrint.getCurrentData ();
            Out << nextValue << " ";
            toPrint.advance ();
        }
    Out << endl;
}
```

- Obiekt *toPrint* jest przekazywany przez referencję, gdyż może być duży i kopiowanie byłoby nieefektywne
- Czy można użyć stałej referencji?
- Co by się stało, gdybyśmy przekazali *toPrint* przez wartość?

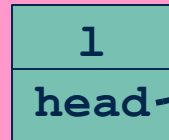
# Przekazywanie obiektów jako parametrów

- # W poprzednim przykładzie obiekt nie może być przekazany przez stałą referencję, gdyż wywołana funkcja zmienia obiekt (wskaźnik *current*)
- # Z tego powodu możemy chcieć wyeliminować możliwość przypadkowej modyfikacji listy i przekazywać ją przez wartość
  - Będzie to rozwiązanie nieefektywne
  - Może spowodować problemy, jeżeli brak konstruktora kopiującego

# Przekazywanie obiektów przez wartość

```
void
PrintList (list toPrint, ostream & Out)
{
    // identyczna implementacja
}
int main()
{
    list BigList;
    // initialize BigList with some data nodes
    PrintList(BigList, cout);
}
```

BigList

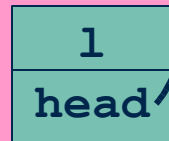


next=0x1234  
val=1

next=0x1334  
val=2

next=NULL  
val=3

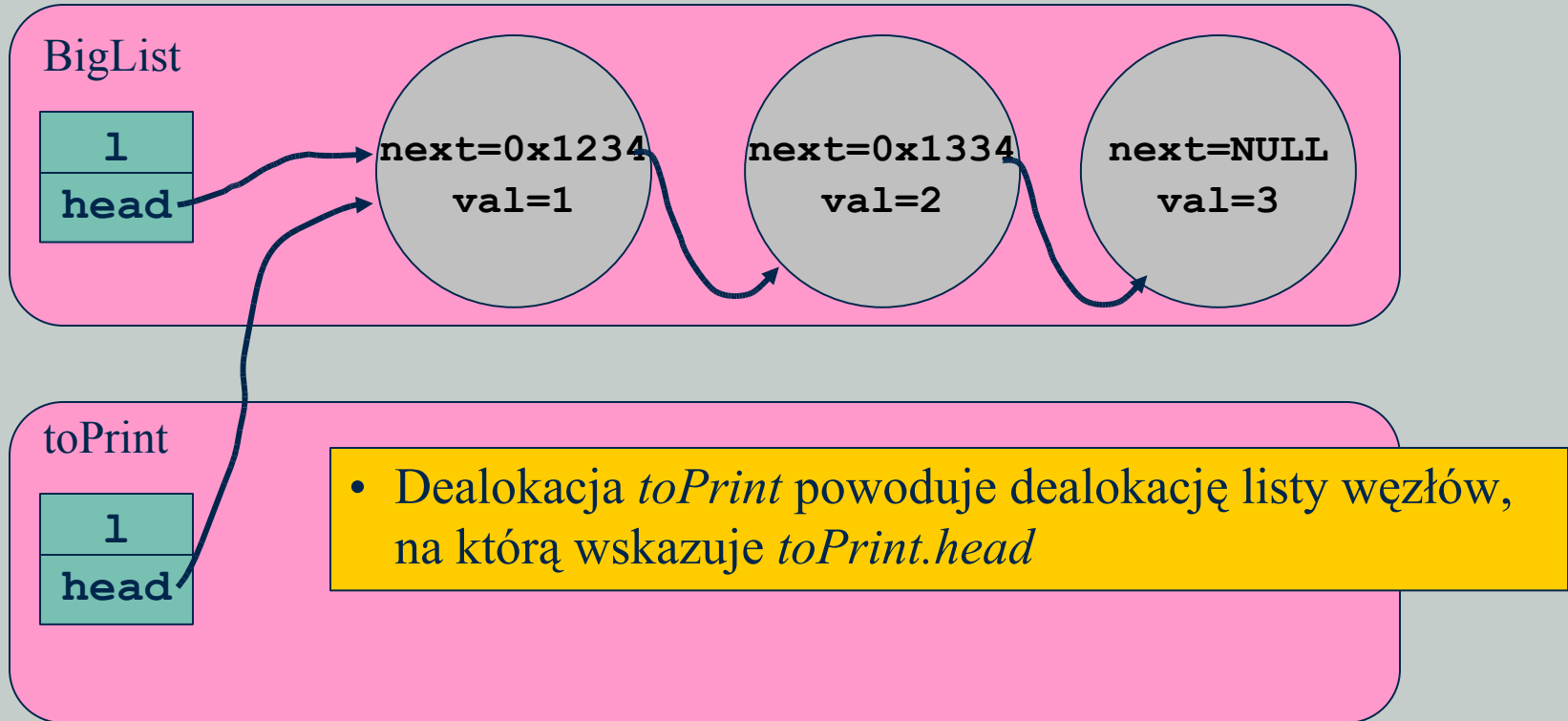
toPrint



- *BigList* i *toPrint* są tym samym obiektem
- Funkcja `PrintList` dalej może modyfikować *BigList*
- Konsekwencje są jeszcze poważniejsze

# Przekazywanie obiektów przez wartość

- ✘ Po zakończeniu *PrintList* czas życia zmiennej *toPrint* kończy się i wywoływany jest jej destruktor



- ✘ Jest to ta sama lista, która jest zawarta w *BigList*. Po powrocie do funkcji *main()*, *BigList* została zniszczona, ale *BigList.head* wskazuje na zwolnioną pamięć

# Przypisanie obiektów

- ⚡ Obiekty posiadają domyślny operator przypisania (identyczny jak struktury)

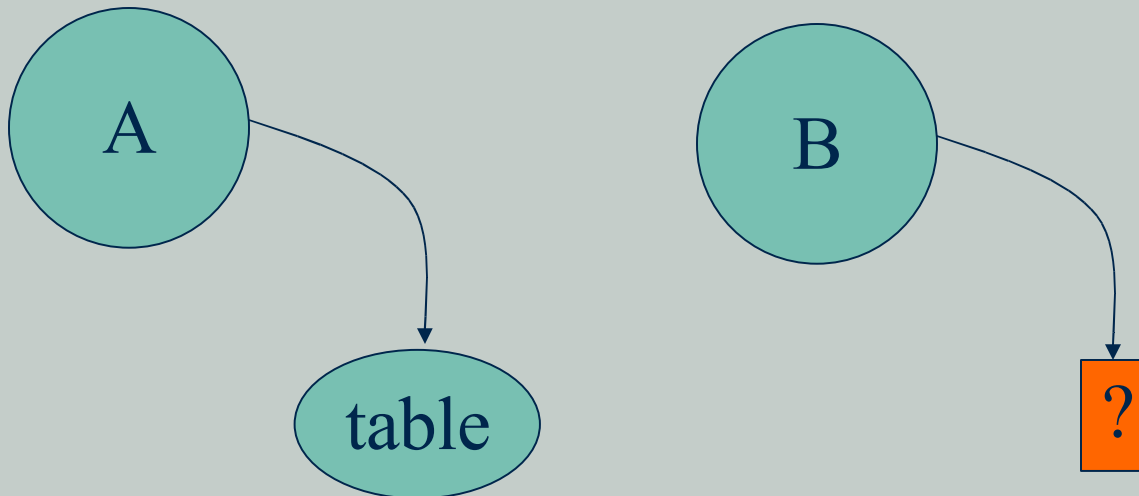
```
class DateType {
    public:
        // constructor
        DateType();
        DateType(int newMonth, int newDay, int newYear);
        ...
};
...
DateType A(1, 22, 2002);
DateType B;
B = A; // copies the data members of A into B
```

- ⚡ Domyślny operator przypisania kopiuje pole po polu wartości z obiektu źródłowego do obiektu docelowego
- ⚡ W wielu przypadkach jest to zadowalające. Niestety, jeżeli obiekt zawiera wskaźnik do dynamicznie zaalokowanej pamięci, rezultat zwykle nas nie zadowoli.

# Problem z przypisaniem wskaźników

```
class Wrong {  
private:  
    int *table; // some data here  
public:  
    // constructor  
    Wrong() { table = new int[1000]; }  
    ~Wrong() { delete [] table; }  
};  
.  
.  
Wrong A;  
Wrong B;  
B = A; // copies the data members of A into B
```

- Jaki typ danych przechowuje *Wrong*?
- Czy *int \* table* to to samo co *int table[]* ?
- Co się stanie przy kopiowaniu?
- Jakie napotkamy problemy?
- Jak temu zapobiec?



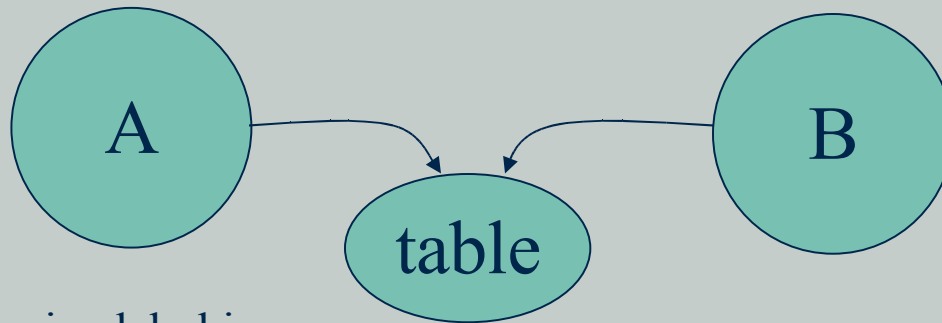


# Kopiowanie płytkie i głębokie

⌘ Wyróżniamy dwa typy kopiowania obiektów zawierających pola będące wskaźnikami

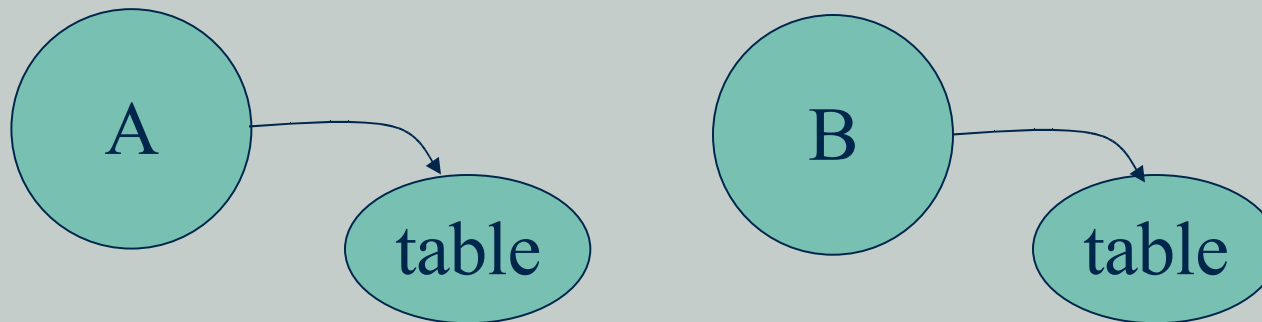
⌘ Kopiowanie płytkie

- Kopiowanie wszystkich składowych (w tym wskaźników)
- Kopiowane są wskaźniki, a nie to, na co wskazują



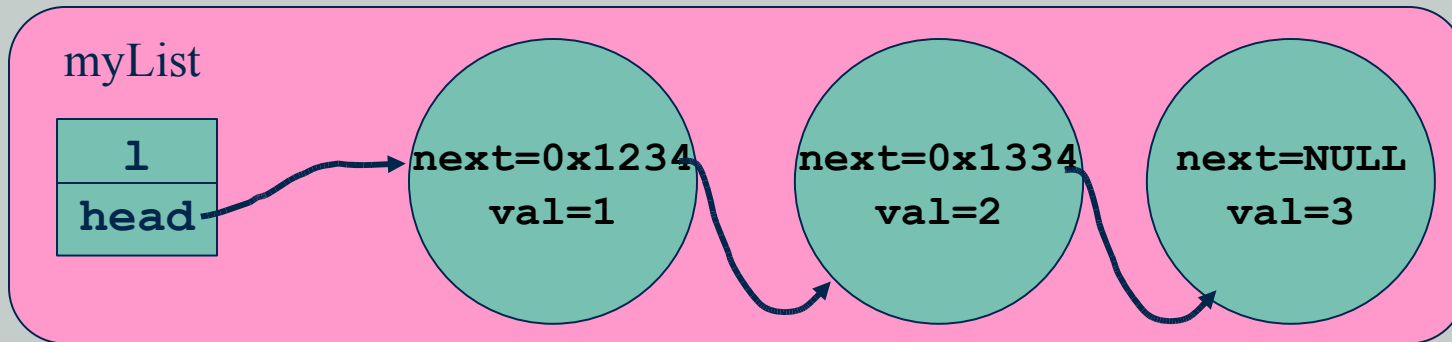
⌘ Kopiowanie głębokie

- Alokacja nowej pamięci dla wskaźników
- Kopiowanie zawartości wskazywanej przez wskaźniki w nowe miejsce
- Kopiowanie pozostałych pól, nie będących wskaźnikami



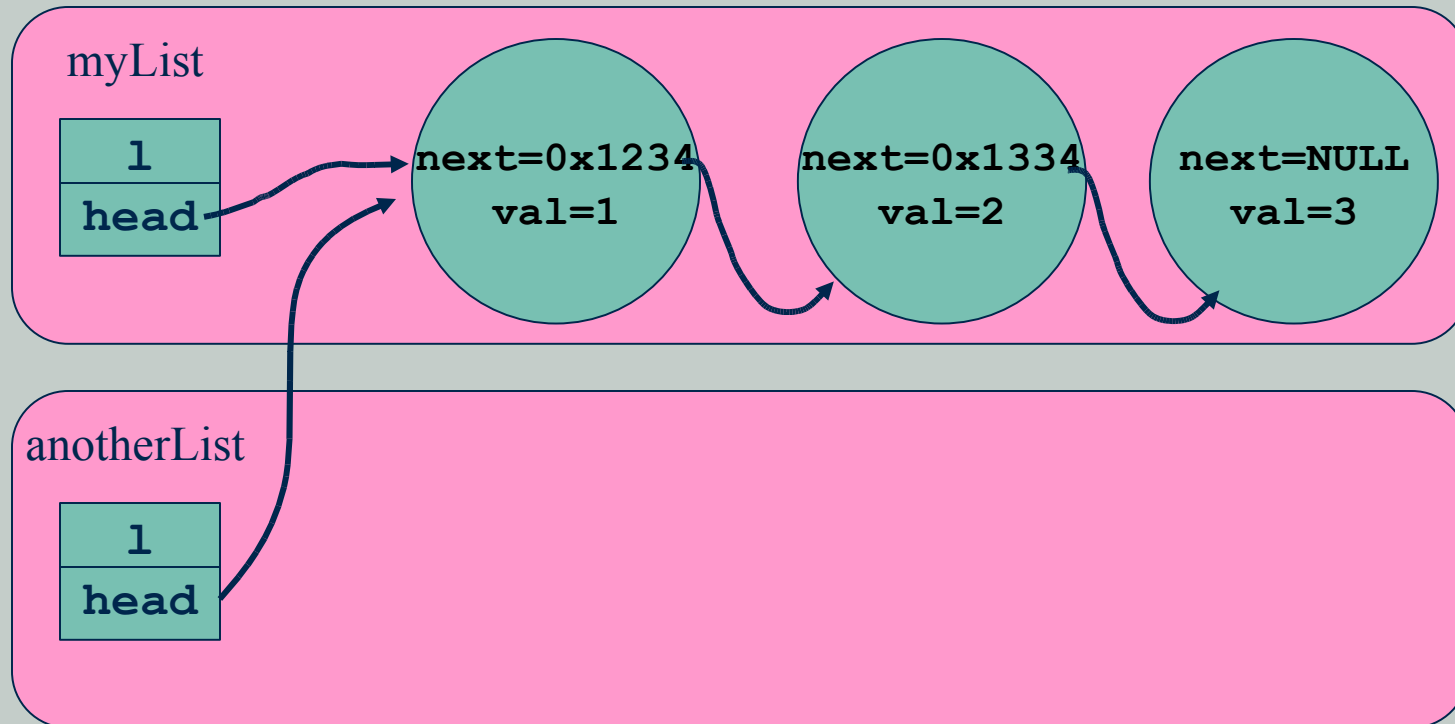
# Problemy z płytkim kopiowaniem

```
list myList;  
myList.insert (3);  
myList.insert (2);  
myList.insert (1);
```



# Problemy z płytkim kopiowaniem

```
list myList;  
myList.insert (3);  
myList.insert (2);  
myList.insert (1);  
  
list anotherList;  
anotherList=myList;
```



# Głębokie kopiowanie

- Kiedy obiekt zawiera wskaźnik do dynamicznie zaalokowanego obszaru, należy zdefiniować operator przypisania wykonujący głębokie kopiowanie
  - W rozważanej klasie należy zdefiniować operator przypisania:  
*AType& AType::operator=(const AType& otherObj)*
  - Operator przypisania powinien uwzględnić przypadki szczególne:
    - Sprawdzić przypisanie obiektu do samego siebie, np. *A=A*:  
*if (this == &otherObj) // if true, do nothing*
    - Skasować zawartość obiektu docelowego  
*delete this->...*
    - Zaalokować pamięć dla kopiowanych wartości
    - Przepisać kopiowane wartości
    - Zwrócić *\*this*

# Konstruktor kopiujący i operator przypisania

- ✚ Konstruktor kopiujący jest używany do stworzenia nowego obiektu
- ✚ Jest prostszy od operatora przypisania - nie musi sprawdzać przypisania do samego siebie i zwalniać poprzedniej zawartości
- ✚ Jest użyty do skopiowania parametru aktualnego do parametru formalnego przy przekazywaniu parametru przez wartość
- ✚ Przy tworzeniu nowego obiektu, można go zainicjalizować istniejącym obiektem danego typu. Wywołany jest wówczas konstruktor kopiujący.

```
int main() {  
    list a;  
    //...  
    list b(a); //copy constructor called  
    list c=a; //copy constructor called  
};
```

# Obiekty anonimowe

⌘ Obiekt anonimowy to obiekt bez nazwy

- Tworzony jest obiekt, ale nie ma nazwanej zmiennej, która go przechowuje

⌘ Użyteczny

- do użytku tymczasowego (parametr przy wywołaniu funkcji, zwracaniu wartości, fragment wyrażenia)
- jako domyślna wartość parametru będącego obiektem

⌘ Rozważmy metodę pobierającą obiekt typu *Address*:

```
void Person::setAddress(Address addr);
```

⌘ Argument może zostać przekazany w sposób następujący:

```
Person joe;  
joe.setAddress(Address("Disk Drive"...));
```

⌘ Zamiast:

```
Person joe;  
Address joeAddress("Disk Drive"...);  
joe.setAddress(joeAddress);
```

# Przykład: obiekty anonimowe jako parametry

## # Bez obiektów anonimowych mamy nieporządek

```
Name JBHName("Joe", "Bob", "Hokie");  
Address JBHAddr("Oak Bridge Apts", "#13",  
"Blacksburg", "Virginia", "24060");  
Person JBH(JBHName, JBHAddr, MALE);  
. . . .
```

## # Użycie obiektów anonimowych zmniejsza zanieczyszczenie lokalnej przestrzeni nazw

```
Person JBH(Name("Joe", "Bob", "Hokie"),  
Address("Oak Bridge Apts", "#13",  
"Blacksburg",  
"Virginia", "24060"), MALE);  
. . . .
```

## Przykład: obiekty anonimowe jako wartości domyślne

- # Użycie obiektów anonimowych jest relatywnie prostą metodą na kontrolowanie inicjalizacji i zmniejszenie liczby metod klasy

```
Person::Person (Name N = Name ("I", "M", "Nobody"),  
Address A = Address ("No Street", "No Number",  
"No City", "No State", "00000"), Gender G =  
GENDERUNKNOWN) {  
    Nom = N;  
    Addr = A;  
    Spouse = NULL;  
    Gen = G;  
}
```



# Wybrane metody tworzenia obiektów

## ☒ Zmienne automatyczne

```
Atype a; //konstruktor domyślny
```

## ☒ Zmienne automatyczne z argumentami

```
Atype a(3); //konstruktor z parametrem int
```

## ☒ Przekazywanie parametrów funkcji przez wartość

```
void f(Atype b) {...}
```

```
Atype a; //konstruktor domyślny
```

```
...
```

```
f(a); //konstruktor kopiujący
```

## ☒ Przypisanie wartości zmiennym

```
Atype a,b;
```

```
...
```

```
a=b; //operator przypisania
```

## ☒ Inicjalizacja nowych obiektów

```
Atype b; //konstruktor domyślny
```

```
...
```

```
Atype a=b; //konstruktor kopiujący (NIE operator przypisania)
```

## ☒ Zwracanie wartości z funkcji

```
Atype f() {
```

```
    Atype a; //konstruktor domyślny
```

```
    ...
```

```
    return a; //konstruktor kopiujący
```

```
}
```

# Cechy dobrze napisanej klasy

## # Jawny konstruktor domyślny

- Gwarantuje, że każdy zadeklarowany egzemplarz obiektu zostanie w kontrolowany sposób zainicjalizowany

## # Jeżeli obiekt zawiera wskaźniki do dynamicznie zaalokowanej pamięci:

### ■ Jawny destruktork

- Zapobiega wyciekowi pamięci. Zwalnia zasoby podczas usuwania obiektu.

### ■ Jawny operator przypisania

- Używany przy przypisywaniu nowej wartości do istniejącego obiektu. Zapewnia, że obiekt jest istotnie kopią innego obiektu, a nie jego aliasem (inną nazwą).

### ■ Jawny konstruktor kopiujący

- Używany podczas kopiowania obiektu przy przekazywaniu parametrów, zwracaniu wartości i inicjalizacji. Zapewnia, że obiekt jest istotnie kopią innego obiektu, a nie jego aliasem.

# Przeciążenie

- # Przeciążenie - istnienie wielu definicji tej samej nazwy
  - Wiele funkcji noszących tę samą nazwę
- # W C++, przeciążone nazwy są rozróżniane na podstawie liczby i typu argumentów
  - z uwzględnieniem dziedziczenia
- # Powyższe parametry zwane są sygnaturą funkcji
  - typy wartości zwracanej nie są rozróżniane, poniższy przykład jest niepoprawny
    - `double fromInt(int x)`
    - `float fromInt(int x)`
- # Częste zastosowanie przeciążenia to przeciążenie operatorów

# Przeciążenie i polimorfizm

- # Przeciążenie to forma polimorfizmu
- # Pozwala na zdefiniowanie nowego znaczenia (funkcjonowania) operatorów dla wybranych typów.
- # Kompilator rozpoznaje, której implementacji użyć, po sygnaturze funkcji (typach operandów użytych w wyrażeniu)
- # Przeciążenie jest wspierane dla wielu wbudowanych operatorów
  - `17 * 42`
  - `4.3 * 2.9`
  - `cout << 79 << 'a' << "overloading is profitable" << endl;`
- # Użyta implementacja zależy od typów operandów

# Przyczyny przeciążania operatorów

- # Wsparcie dla naturalnego, sugestywnego użycia:
  - `Complex A(4.3, -2.7), B(1.0, 5.8);`
  - `Complex C;`
  - `C = A + B;` // '+' oznacza dodawanie dla tego typu, tak samo jak dla np. typu `int`
- # Integralność semantyczna: przypisanie dla typów obiektowych musi zapewnić wykonanie głębokiej kopii
- # Możliwość użycia obiektów w sytuacjach, w których oczekiwany jest typ prosty

# Operatory, które mogą być przeciążane

# Jedynie poniższe operatory mogą być przeciążane

+	-	*	/	%	^	&
	~	!	=	<	>	+=
-=	*=	/=	%=	^=	&=	!=
<<	>>	>>=	<<=	==	!=	<=
>=	&&		++	--	->*	,
->	[]	()	new	new[]	delete	delete []

# Operatory =, ->, [], () muszą być metodami niestatycznymi

# Wskazówki dotyczące przeciążania operatorów

- # Operator powinien zachowywać się zgodnie z oczekiwaniami użytkownika

```
Complex Complex::operator~() const {  
    return ( Complex(Imag, Real) );  
}
```

- # Należy dostarczyć pełen zestaw pokrewnych operatorów:  $a = a + b$  i  $a += b$  powinny mieć taki sam efekt i należy dostarczyć użytkownikowi klasy oba operatory
- # Należy zdefiniować operator jako składową jeśli inne rozwiązanie nie jest konieczne
- # Jeżeli przeciążony operator nie może być składową, powinien być funkcją zaprzyjaźnioną, a nie używać akcesorów, dodanych do klasy wyłącznie w tym celu

# Składnia przeciążania operatorów

⚡ Deklarowane i definiowane tak samo jak inne metody i funkcje, różnią się użyciem słowa kluczowego *operator*

⚡ Jako metoda klasy:

```
bool Name::operator==(const Name& RHS) {  
    return ((First == RHS.First) &&  
            (Middle == RHS.Middle) &&  
            (Last == RHS.Last) );  
}
```

⚡ Jako funkcja zaprzyjaźniona:

```
bool operator==(const Name& LHS, const Name& RHS) {  
    return ((LHS.First == RHS.First) &&  
            (LHS.Middle == RHS.Middle) &&  
            (LHS.Last == RHS.Last) );  
}
```

⚡ Bardziej naturalne jest użycie metody



# Użycie operatorów przeciążonych

# Jeżeli *Name::operator=* jest zdefiniowany jako metoda klasy *Name*, wówczas

```
nme1 == nme2
```

jest równoważne

```
nme1.operator==(nme2)
```

# Jeżeli *operator==* nie jest zdefiniowany jako składowa, wówczas

```
nme1 == nme2
```

jest równoważne

```
operator==(nme1, nme2)
```

# Przeciążenie operatorów

- Składowe przeciążające operatory są definiowane z użyciem słowa kluczowego *operator*

```
// add to DateType.h:  
bool operator==(Datetype otherDate) const ;
```

Interfejs

```
// add to DateType.cpp:  
bool dateType::operator==(Datetype otherDate) const  
{  
    return( (Day == otherdate.Day ) &&  
            (Month == otherDate.Month ) &&  
            (Year == otherDate.Year ) );  
}
```

Implementacja

```
DateType aDate(10, 15, 2000);  
DateType bDate(10, 15, 2001);  
if (aDate == bDate) { . . .
```

Klient

- Odpowiednio użyte przeciążenie operatorów pozwala na traktowanie obiektów typu zdefiniowanego przez użytkownika w sposób tak samo naturalny, jak typów wbudowanych.

# Operator binarny jako składowa

- # Operator odejmowania dla klasy *Complex* jako składowa:

```
Complex Complex::operator-(const Complex& RHS) const {  
    return ( Complex(Real - RHS.Real, Imag - RHS.Imag) );  
}
```

- # Lewy operand operatora **musi** być obiektem

```
Complex X(4.1, 2.3), Y(-1.2, 5.0);  
int Z;
```

OK: X + Y;

Not OK: Z + X;

- # Zazwyczaj przekazuje się operand przez stałą referencję, żeby uniknąć narzutu kopiowania

# Operator binarny jako funkcja nie będąca składową

- Operator odejmowania dla klasy *Complex* jako funkcja nie będąca składową:

```
Complex operator-(const Complex& LHS, const Complex& RHS) {  
    return ( Complex(LHS.getReal() - RHS.getReal(),  
                    LHS.getImag() - RHS.getImag()) );  
}
```

- Operator odejmowania musi używać interfejsu publicznego w celu dostępu do danych prywatnych klasy...
  - ...chyba że klasa *Complex* zadeklaruje funkcję jako funkcję zaprzyjaźnioną
- Funkcja zaprzyjaźniona ma dostęp do składowych prywatnych klasy tak samo, jak składowa.

```
class Complex  
{  
    ...  
    friend Complex operator+ (const Complex&, const Complex&);  
    ...  
};
```

# Operatory jednoargumentowe (unarne)

## # Operator negacji dla klasy *Complex*

```
Complex Complex::operator-() const {  
    return ( Complex(-Real, -Imag) );  
}
```

```
Complex A(4.1, 3.2); // A = 4.1 + 3.2i  
Complex B = -A; // B = -4.1 - 3.2i
```

# Składowa implementująca unarny operator nie ma parametrów.

# Operator pre- i postinkrementacji

```
class Value {
private:
    int x;
public:
    Value(int i = 0) : x(i) {}
    int get() const { return x; }
    void set(int x) ( this->x = x; )
    Value& operator++();
    Value operator++(int Dummy);
}
```

## Operator preinkrementacji

```
Value& Value::operator++() {
    x = x + 1;
    return *this;
}
```

## # Operator postinkrementacji

```
Value Value::operator++(int Dummy) {
    x = x + 1;
    return Value(x-1); // return previous value
}
```

# Wielokrotne przeciążenie

- # W klasie może być kilka operatorów dodawania

```
Complex Complex::operator+(double RHS) const {  
    return (Complex(Real + RHS, Imag));  
}  
Complex Complex::operator+(Complex RHS) const {  
    return (Complex(Real + RHS.Real, Imag + RHS.Imag));  
}
```

- # Napiszmy kilka wyrażeń mieszanych:

```
Complex X(4.1, 2.3);  
double R = 1.9;  
Complex Y = X + R; // Y.Real is 6.0
```

- # Sygnatura funkcji zostanie użyta do wyboru właściwej funkcji

```
Complex Z = Y + R; // complex plus double  
Complex W = Y + X; // complex plus complex
```

# Wielokrotne przeciążenie

## # Konstruktor może służyć jako operator konwersji

```
Complex Complex::operator+(Complex RHS) const {  
    return (Complex(Real + RHS.Real, Imag + RHS.Imag));  
}  
Complex::Complex (double co)  
{  
    Real = co;  
    Imag = 0;  
};
```

```
Complex X(4.1, 2.3);  
double R = 1.9;  
Complex Y = X + R; // Y = X.operator+(Complex(R));
```

## # Nie zadziała, gdy lewy argument jest typu double

```
Complex X(4.1, 2.3);  
double R = 1.9;  
Complex Y = R + X; // syntax error
```

## # Lepiej zdefiniować operator dwuargumentowy jako funkcję zaprzyjaźnioną



# Wielokrotne przeciążenie

- ⌘ Funkcja zaprzyjaźniona działa również, gdy argument typu *double* jest po lewej stronie

```
friend Complex operator+(Complex LHS, Complex RHS) {  
    return (Complex(LHS.Real + RHS.Real, LHS.Imag + RHS.Imag));  
}
```

```
Complex X(4.1, 2.3);  
double R = 1.9;  
Complex Y = X + R; // Y = operator+(X, Complex(R));  
Complex Z = R + X; // Y = operator+(Complex(R), X);
```

- ⌘ Kiedy implementować operatory jako funkcje zaprzyjaźnione:

- Przy operacjach na podstawowych typach danych,  
np. *Complex operator+(int LHS, const Complex& RHS);*
- Kiedy nie można zmodyfikować klasy oryginalnej,  
np. *ostream*

# Implementowany zbiór operatorów

- # W wielu przypadkach dla danego typu cała kategoria operatorów ma sens
- # Na przykład, dla klasy *Complex* ma sens przeciążenie wszystkich operatorów arytmetycznych. Dla klasy *Name* ma sens przeciążenie wszystkich operatorów relacji
- # Często implementacje jednych operatorów mogą korzystać z implementacji innych operatorów,

np:

```
Complex operator + (Complex s1, Complex s2)
{
    Complex n (s1);
    return n += s2;
}
```

# Operatory wejścia/wyjścia

- ✘ Nie mamy dostępu do kodu klas *istream* i *ostream*, a więc nie możemy przeciążyć operatorów `<<` i `>>` jako składowych tych klas.
- ✘ Nie możemy ich także uczynić składowymi klasy danych, ponieważ obiekt klasy danych musiałaby wówczas znajdować się po lewej stronie operatora, a nie o to nam chodzi.
- ✘ Dlatego musimy zdefiniować *operator<<* jako osobną funkcję
- ✘ Funkcja ta musi mieć dostęp do składowych klasy danych, a więc zazwyczaj będzie to funkcja zaprzyjaźniona. Rozwiązaniem alternatywnym, często nieakceptowalnym, jest zdefiniowanie akcesorów dla wszystkich pól prywatnych.
- ✘ Sygnatura funkcji będzie miała następującą postać:  

```
ostream& operator<<(ostream& Out, const Data& toWrite)
```

# *operator*<< dla obiektów *Complex*

- # Przeciążony *operator*<< drukuje sformatowany obiekt klasy *Complex* do strumienia wyjściowego

```
ostream& operator<<(ostream& Out, const Complex& toWrite) {  
    const int Precision = 2;  
    const int FieldWidth = 8;  
    Out << setprecision(Precision);  
    Out << setw(FieldWidth) << toWrite.Real;  
    if (toWrite.Imag >= 0)  
        Out << " + ";  
    else  
        Out << " - ";  
    Out << setw(FieldWidth) << fabs(toWrite.Imag);  
    Out << "i";  
    Out << endl;  
    return Out;  
}
```

## *operator*>> dla obiektów *Complex*

# Przeciążony *operator*>> wczytuje ze strumienia wejściowego obiekt klasy *Complex* sformatowany w sposób używany przez *operator*<<

```
istream& operator>>(istream& In, Complex& toRead) {  
    char signOfImag;  
    In >> toRead.Real;  
    In >> signOfImag;  
    In >> toRead.Imag;  
    if (signOfImag == '-')  
        toRead.Imag = -toRead.Imag;  
    In.ignore(1, 'i');  
    return In;  
}
```

Funkcja zależy od sposobu formatowania obiektów *Complex* w strumieniu wejściowym. Może być znacznie bardziej skomplikowana, jeżeli chcemy wczytywać różne formaty.

# Przeciążenie operatora indeksowania

```
class vector
{
    int *dane;
    unsigned int size;
public:
    vector(int n); //creates n-element vector
    ~vector();
    int& operator[] (unsigned int pos);
    int operator[] (unsigned int pos) const
    //copy constructor, assignment operator, ...
};
int& vector::operator[] (unsigned int pos)
{
    if (pos >= size)
        abort ();
    return dane[pos];
}
int vector::operator[] (unsigned int pos) const
{
    if (pos >= size)
        abort ();
    return dane[pos];
}
```

■ Dostarcza spodziewaną funkcjonalność, pozwalając na napisanie

```
vector a(10);
a[5]=10;
cout << a[4]<<endl;
```

# Przeciążenie operatorów relacji

- # Jeżeli zamierzamy przechowywać obiekty danej klasy jako elementy kolekcji, przynajmniej część operatorów relacji powinna zostać przeciążona
- # W celu efektywnego wyszukiwania elementów i sortowania kolekcja musi mieć możliwość porównywania przechowywanych obiektów. Może ona:
  - Użyć funkcji akcesorów i porównywać bezpośrednio pola
  - Użyć specjalnej metody porównującej zdefiniowanej w danej klasie
  - Użyć przeciążonych operatorów relacji (`==`, `!=`, `<`, `<=`, `>`, `>=` w zależności od potrzeb)
- # Pierwsze podejście wymaga, aby kolekcja posiadała szczegółowe informacje o swoich elementach
- # Drugie podejście wymaga dostarczenia specjalnych składowych
- # Trzecie podejście jest najbardziej naturalne i pozwala na niezależne projektowanie klas danych i kolekcji.