# References

Reference is possible to consider as an implicit pointer. It can be used:

- as a parameter of the function,
- as a value returned by the function,
- as a reference variable.

- References as a function parameter. Parameters can be passed to a function in three ways:

  - by value - a copy of the argument is passed to the function.
  - by pointer - a pointer to the argument is passed.
  - by reference – an implicit pointer (reference) to the argument is passed.

- A pointer to an argument in C++ can be passed explicitly (just like in C), and implicitly - a reference parameter can be used.

To declare the reference parameter before the name of the formal argument, put *&*.

In the function body, we do not write * before the name of the formal argument (*indirection operator*). Statement *n* = 100; means that the number 100 is assigned to the actual argument, which replaces the formal argument n when calling the function.

When calling the function, we will not put *&* before the actual parameter. When declaring the function, it is declared that the formal argument is a reference parameter - the compiler automatically passes the address of the actual argument to the function.

The reference parameter is not a pointer - pointer arithmetic doesn't work here:

```
int k = 100;
fun(k);
void fun(int &n)
{
      n++;            //increases n by one.
      n = n+20;       // the value of n will be
                      // increased by 20
}
```

```
int k = 100;    fun(&k);
void fun(int *n)
{
      n++;             //  n points to the address n+sizeof(int)
      n = n+20;       //n points to the address
                       //n + 20*sizeof(int)
      n = n – 21;     //points to the k
      *n = *n+20;  //increasing the variable with the
                      //address n by 20
}
```

**Passing objects to functions by references**

Passing an object as an argument (by value) creates a copy of that object - calling the destructor at the end of the function job - possible side effects, time to call the function is relatively long due to copying of arguments. Changes to the data of the object, made in the function, take place in the copy, after the end of the function, they do not appear in the original object.

Passing an object by reference is not created its copy. After the function completes, the object passed as a parameter is not destroyed - the destructor is not called. Passing is faster than passing by value (missing a copying of object). Changes in the object data, made in the function, after the function ends, appear in the original object. The syntax is the same as that for passing by value:

```
name_of_reference_to_object.variable
name_of_reference_to_object.class_method(list of arguments);
```

**Example W16.**

## The function returns a reference

Functions can return references. This makes it possible to have this function on the left side of the assignment operator and also is advantageous in overloading operators.

**Example W17.**

# The overloading of operators

Operator overloading allows you to use the same syntax for class objects and structures as for standard object types.

You can overload an operator by creating operator functions. The operator functions determine what operations the operator should perform on objects of the specified class. The keyword *operator* is used to create an operator function. An operator function can be a member of a class (and it might not be). Most often, an operator function that is not a class member is a friend of that class.

The operator function is a conventional function but has a specific calling interface. For instance, in the code line:

```
my_class a(….), b(….);
a = b;
a = a + b;
```

compiler automatically calls to the operator =, if class *my_class* comprises the overloading of operator function = . Otherwise (if does not comprise), compiler applies the bitwise copy.

## Operator function - class member

```
typ  class_name :: operator  #  (list_of_arguments)
{
      //function body
}
```

Often the return value type of an operator function is the class for which that function is designated. The # sign indicates the operation of the operator we are overloading.

## Restrictions on operator overload:

- The priority of operators must not be changed.
- The number of the operator's operands must not be changed.
- Operators must not be overloaded:  .  ::  ->  ?
- Operators - functions cannot have default parameters.

Operator - functions, except the = operator, are inherited from the derived class.

For a derived class, you can overload any operator, even those operators that were already overloaded in the base class.

In order to properly overload the operator, you need to clearly understand the following:

- how many operands operator contains.
- what type of object the operator function returns.
- what type is each of the operands.
- Is it possible to overload the operator as a member function of the class, or is it necessary to overload it as a separate function and make it friends with the class.

## Overload of binary operators

- Such an operator has two operands.
- The operator – function class member has only one parameter - reference or pointer to the right side operand. The left operand is passed by pointer *this*.

# Overload of the assignment operator

It is a binary operator - it has two operands, left and right.

Example:

```
coord & coord::operator = (const coord &right)
/*================================================
Overload of the assignment operator: returns the reference to object
================================================*/
{
   x = right.x;
   y = right.y;
   return *this;
}
```

Returns a reference to an object - the left-hand operand is L-value. In addition, when assigned, left operand it changes. The function returns * *this* - gets the value through this pointer and returns a reference. This makes it possible to use the chain a = b = c;

This means that first the object b will be assigned the values of the corresponding components of the object c (b = c), and then - the values of the components of the object b will replace the corresponding values of the components of the object a (a = b).

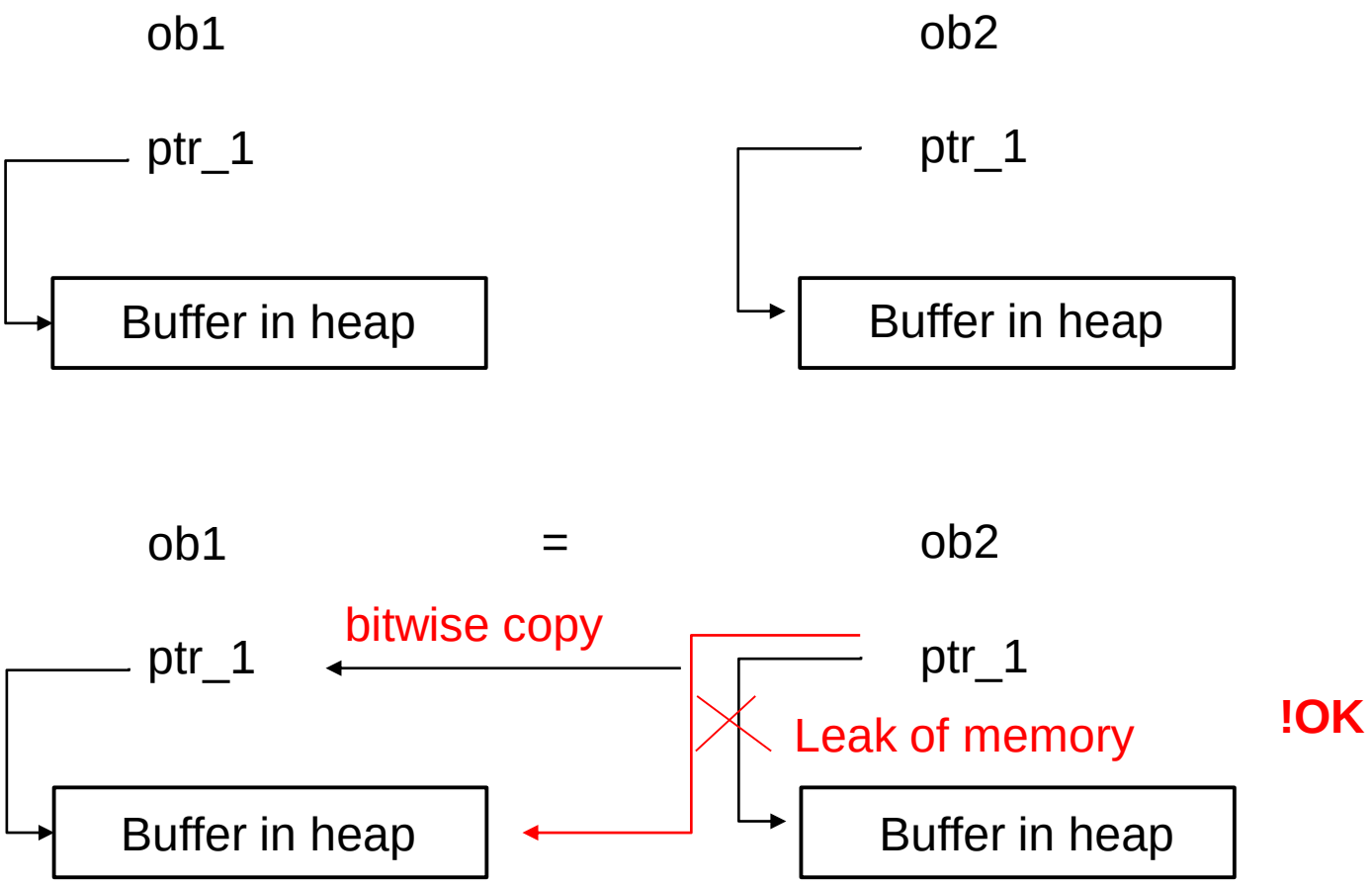
The parameter operator-function ”=” is a reference to the object - the right operand. Passing the reference parameter to function does not create a copy of the object - it is significantly faster than passing the object and does not cause side effects.

## Assignment of objects

- If the type of two objects is the same, one object can be assigned to the other. By default, assigning one object to another means that the data of the object on the right will be copied to the object on the left bit by bit.

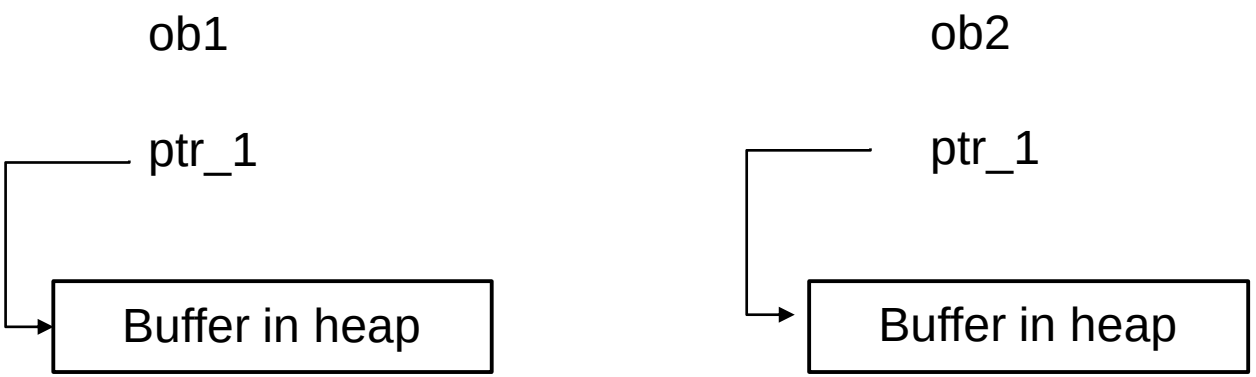
```
class myclass
{
        int i;
public:
        myclass(int ii) : i(ii) { }  //It is similar to: myclass(int ii) { i = ii; }
        myclass( ) : i(0) { }
        int get_i() { return i; };
};

int main()
{
        myclass ob1(10), ob2;
        ob2 = ob1;
        cout << ob1.get_i() << ”\t” << ob2.get_i() << endl;
……..
}

        Output:
        10      10
```

➢ If the class data contains pointers to objects of any type, the default assignment may cause an error.

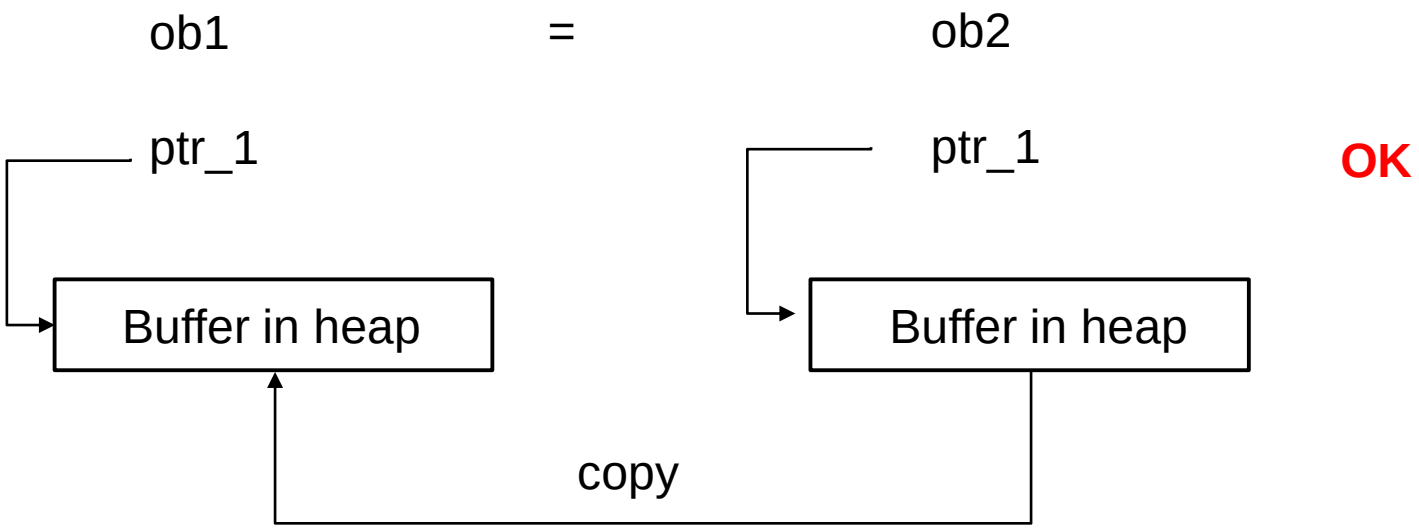



This buffer in heap is shared between the both objects: ob1 and ob2.

- To correct such a situation, it is necessary to overload the operator = .




Assign operator = overload




**OK**

The = operator overload allows on correct solution of such a problem.

**Example W13**

# Overload of operators + - * \

**Example W18**

```
class coord
{
      double x, y;
public:
      . . . . . .
};
```

```
coord coord::operator + (const coord &right) const
{
      coord ret;
      ret.x = x+right.x;
      ret.y = y+right.y;
      return ret;
}
```

Function-operator + returns the *coord* type object. Inside this function is created an auxiliary object *ret*. It possible to use this operator in complex expressions as a + b + c, for the fact that a + b + c = (a + b) + c and the first sum must be stored in the memory so as not to change any of the terms. We use function that returns an object. We pass the right operand by reference to speed up execution and avoid creating a copy of the object, like when passing the object itself and then destroying that copy by calling a destructor.

```
double coord::operator * ( const coord &right) const
/*=================================================================
dot_prod
=================================================================*/
{
      double ret = 0.0;
      ret = x*right.x+y*right.y;
      return ret;
}
```

The result of the dot product is the real value - operator-function returns a double value. The first vector (the left-hand operand) is taken by the pointer *this*, and the right-hand operand is passed as the reference parameter.

```
coord coord::operator * (const double alpa) const
/*=================================================================
mnozenie wektora przez skalar alpa
=================================================================*/
{
      coord ret;
      ret.x = x*alpa;
      ret.y = y*alpa;
      return ret;
}
```

The result is a vector (object of *coord* type) and the scalar is passed as a parameter.

The left operand must be an object of the class *coord* . The right operand must be scalar. Conversely (the left operand - scalar) is impossible because scalar is not the object of the class *coord*. The operator * generates a call to the operator-function and passing the left operand by pointer *this* and right operand - as an actual argument of the function-operator.

## Overloading of relational and logical operators

Relational and logical operators return 0 (*false*) or !0 (*true*), so the return type can be *int* or *bool*.

Relational and logical operators can meet in complex expressions that contain data of other types.

**Example W19.**

*The overload of the == operator regarding the coord class.*

```
int coord::operator == (const coord &right) const
{
        return (x == right.x && y == right.y);
}
```

The left-hand operand is an object of the class *coord*, generates an operator-function call, and is passed by *this*. The right-hand operand is also a *coord* object and passed by reference. Class data are the components of a vector, two vectors are equal if each of the respective components is equal.

*The > operator overload regarding the coord class*.

```
int coord::operator > (const coord &right) const
{
   double ro_left, ro_right;
   ro_left  = x*x+y*y;
   ro_right = right.x* right.x+ right.y* right.y;
   return (ro_left > ro_ right);
}
```

We define for an object of a given class the relations >, <, >=, <= are determined on the basis of comparing the modulus of these vectors (the norm || ... $||_2$ of the vector). This does not mean that such a definition is unambiguous. It could be compared on the basis of the norm || ... $||_\infty$ . So, it depends on which norm for a given case we consider more representative.

# Friendly operator-functions

For friend functions, *this* pointer is not passed - we need to pass both the left operand and the right operand (for binary operators) explicitly.

The other feature of operator overload for friendly functions remains the same as for operator-member functions of the class.

For overloading the assignment operator (=), and also the operator [ ], only the operator - class member functions can be used (operator - class-friendly functions cannot be used).

Friendly operator-function declaration for the class:

```
friend typ_ret_val operator # (typ_left  left_operand, typ_right  right_operand);
```

Definition of the function-operator - not a class member (binary operator):

```
friend typ_ret_val operator # (typ_left  left_operand, typ_right  right_operand)
{
  . . . . . .  //body
}
```

In the argument list, the left operand is first and the right operand is second.

For class member function-operator, the left operand is passed implicitly by *this* pointer - that is, the left operand must be the type of the class object.

## Overload extraction – insertion operators

```
istream & operator >> (istream & stream, typ_class & ob)
{
        //extractor's body
        return stream;
}
```

The operator >> should be included in the chain *mystream* >> *object*, for this it returns a reference to the stream *mystream*. The >> operator has two operands - a stream and references to a class object, the left operand is a stream, and the right operand is a reference to a class object. Therefore, it is impossible to overload operator >> as a class-member function-operator.

## Example (project_1):

```
class mcoord    //base class
{
protected:
   double x;
   double y;
public:
   mcoord(double xx, double yy) { x = xx; y = yy; }
   mcoord() {x=0; y=0; }
};

class node : public mcoord     //derived class
{
   int numb;
   char str[512];
public:
   node(int nb, char *st, double xx, double yy);
   ~node() { }
……..
   friend ostream & operator << (ostream &stream, const node &ob);
   friend istream & operator >> (istream &stream, node &ob);
   friend ostream & operator << (ostream &stream, const node * ob);
   friend istream & operator >> (istream &stream, node * ob);
};
```

## Formatted IO

```
istream & operator >> (istream &stream, node &ob)
/*==========================================================
Extractor – overload of operator >> for an object of a node class
==========================================================*/
{
   stream >> ob.x >> ob.y >> ob.numb >> ob.str;
   return stream;
}

ostream & operator << (ostream &stream, const node &ob)
/*==========================================================
Inserter – overload of operator << for an object of a node class
==========================================================*/
{
   stream << ob.x << "  " << ob.y << "  " << ob.numb << "  " <<  ob.str << endl;
   return stream;
}
```

## Unformatted (binary) IO

```
istream & operator >> (istream &stream, my_node *ob)
{
   stream.read((char*)ob, sizeof(*ob));
   if (stream.bad())
   {
         //call error handler
         MY_DATA_GLOBAL::msg.mess(ERR_ACCESS_FILE);
   }
   return stream;
}

ostream & operator << (ostream &stream, const my_node *ob)
{
   stream.write((const char*)ob, sizeof(*ob));
   if (stream.bad())
   {
         //call error handler
         MY_DATA_GLOBAL::msg.mess(ERR_ACCESS_FILE);
   }
   return stream;
}
```

## Array index operator [ ] overload.

In C ++, operator [ ] is a binary operator, and it can only be overloaded as a class member.

```
return_typ  & class_name :: operator  [ ]  (int index)
{
    //……..
}
```

The [ ] operator returns a reference to the element of array with the given index. This makes it possible to use ob [i] =… on the left side of the assignment operator. Left operand is a class object, right operand is an index of the array element.

**Example W20.**

The safe array requires a significant amount of work, which causes a considerable decrease in the calculation performance. For this, in C ++, the boundaries of arrays are not controlled. In the debug version you can use the safe array, and in the release version - remove such a control.