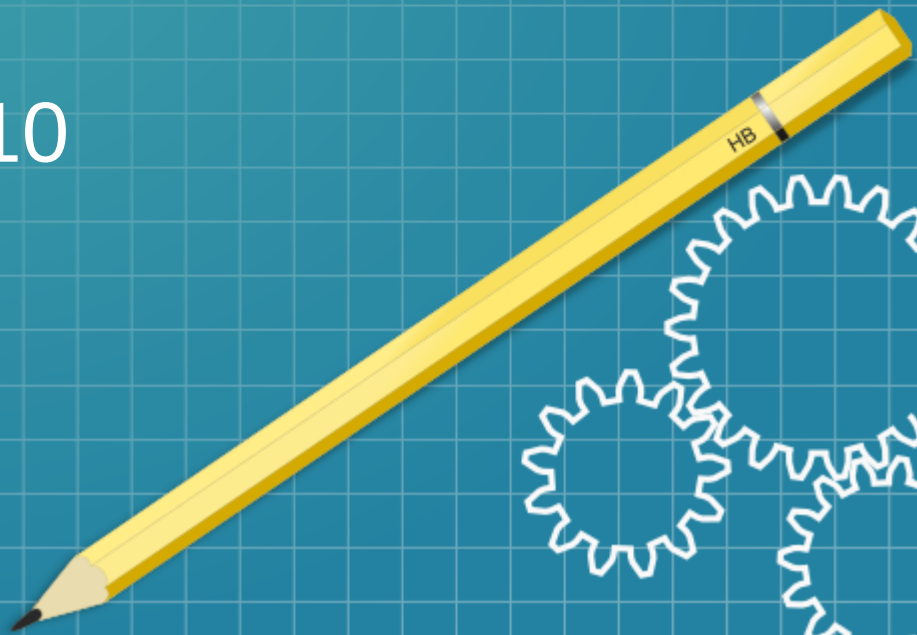
Operator overloading
Class `string`
(part 2)

Chapter 10



Today we will

We will work on the `Array class` and along the way we will discuss

- Dynamic memory management
- Destructors
- Copy constructors
- Overloading operators as member functions and as non-member functions

Dynamic Memory Management

Consider the following code fragment:

```
int *x, *y, *z;
x = new int;
*x = 3;
y = new int;
*y = 4;
z = x;
x = y;

delete z;
delete y;
```

`new` statement allocates dynamic memory and returns the starting address.

`delete` statement deallocates memory that was dynamically allocated.

see dynamicMemoryAllocation1.cpp

Dynamic Memory Management

Remember: each `new` statement that is executed must eventually have a corresponding delete statement that is executed to deallocate the memory.

If you forget a `delete` statement, your program will have a memory leak. Even though a program with *memory leak* may not crash, the code is considered incorrect.

Dynamic Memory Management

Dynamic Arrays

A *dynamic array* is explicitly declared as a pointer:

```
int *a;
```

It is given an initial size using the new operator:

```
a = new int[5];
```

A *dynamic array can be expanded*: its items can be copied into a larger area, whose address can be assigned to the original variable.

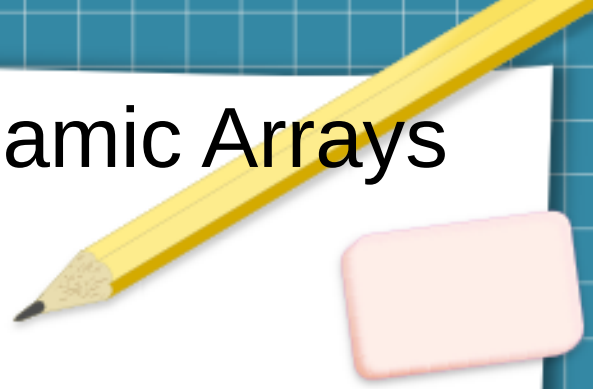
Dynamic Memory Management: Dynamic Arrays

Consider the following code fragment:

```
int *data, *temp, i;

data = new int[5];

for (i=0; i<5; ++i) {
    data[i] = i; }

temp = new int[10];

for (i=0; i<5; ++i) {
    temp[i] = data[i];}

delete [] data;
data = temp;

for (i=0; i<10; ++i) {
    data[i] = i; }
delete [] data;
```

see dynamicMemoryManagement2.cpp

Dynamic Memory Management: Dynamic Arrays



Starting from C++ 11, there is a “smart pointer” `unique_ptr` for managing dynamically allocated memory.

When a `unique_ptr` goes out of scope, its destructor automatically returns the managed memory fo the free store.

We will use it in Chapter 17.

class Array

Let's define a class Array, which will be

- a *fixed size* array
- with all elements of type *int*
- the space for the array will be allocated dynamically
- will have a *copy constructor*

  `Array a(b)` or `Array a{b}`
- will have comparison for equal/not-equal `a == b`
- will have indexing/subscript operator [ ] `a[5]`
- will have the assignment operator `a = b`

See array.h, array.cpp and testingArray.cpp

Operator Overloading summary

- C++ *does not allow new* operators to be created, but
- it *does allow most* existing operators to be overloaded
- when operators are overloaded as member-functions, they must be *non-static* (since they will be called on a object of a class)
- operators that cannot be overloaded: . .* :: ?:

pointer to member

- an *operator's precedence* cannot be changed by overloading (we can use parentheses for *force* the order of evaluation)
- an *operator's associativity* cannot be changed by overloading

Operator Overloading summary

- an *operator's "arity"* (number of operands) cannot be changed by overloading
- we *cannot overload* operators to change how an operator works on fundamental-type values, i.e.
- operator *overloading works only with user-defined types or with a mixture of an object of user-defined type and an object of fundamental type*.

Operator Overloading summary

- related operators, like + and +=, must be overloaded separately
- when overloading ( ), [ ], → or any other assignment operator, the overloading function *must be declared as a class member*
- for all other overloadable operators, the operator overloading functions can be *member functions* or *non-member functions*

Member vs Non-Member Functions

Recall the equality operator overloading for the `class Array`:

```
class Array {
   ...
public:
   ...
   bool operator==(const Array&) const;
   ...
```

It is overloaded as a member function

Member vs Non-Member Functions

We could also overload it as a non-member function:

```
class Array {
   ...
public:
   …
};

bool operator==(const Array&, const Array&);
```

In some cases, we need to announce them as `friend` functions in order to have access to the attributes of the class

Member vs Non-Member Functions

Overloaded operator functions can be member functions *only* when the *left* operand is an object of the class in which the function is a member.

Recall that we overloaded the `operator>>` and the `operator<<` as non-member, friend functions.

Unwanted Member Functions

Sometimes, we want to prohibit some operations on object of a class, for example, *copy constructor*, or *assignment operator*.

In this case we can:

- declare them as `private`
- starting from C++ 11: *`delete`* them from our class:

```
Array(const Array&) = delete;
```

or

```
const Array& operator=(const Arrya&) =
delete;
```

Overloading Function Call Operator()

Consider this code fragment:

```
String String::operator()(size_t startIndex,
size_t endIndex) const {

// check the range
// return the sub-string starting from
// position startIndex, and ending with
// position endIndex, including
}
...
String st1="Social"
st1(2,4)  // generates call st1.operator(2,4)
```

returns “cia”

HW assignment

2) add the following to the `class Complex`:
(a) overload the *input stream operator* to get the real and the imaginary parts of a complex number (`cin << a`)

(b) overload the `operator==` , the comparison for equality of two complex numbers (do it as member method)

```
bool operator==(const Complex& other) const;
```

(c) overload the `operator!=` , the comparison for not-equal of two complex numbers (do it as member method)

```
bool operator!=(const Complex& other) const;
```

(d) overload the `+` , `-`, `/`, and `*` operators, as member methods

```
Complex operator+(const Complex& other) const;
Complex operator-(const Complex& other) const;
...
```

HW assignment

Self-Study:

10.12 *Converting Between Types*

10.13 `explicit` *Constructors and Conversion Operators*

Suggested exercises

(*not for grade, but the questions related to these will appear on a quiz or a test*):

2) Chapter 10, Exercises 10.10 and 10.11