

```
class Polynomial {
public:
    Polynomial();
    Polynomial(const string& N, const vector<int>& C);
    Polynomial operator+(const Polynomial& RHS) const;
    Polynomial operator-(const Polynomial& RHS) const;
    Polynomial operator*(const Polynomial& RHS) const;
    Polynomial Derivative() const;
    Polynomial Antiderivative() const;
    int Integral(int a, int b) const;
    int operator()(int x) const;

    string getName() const;
    void setName(const string& N);
    ~Polynomial();

private:
    string Name;
    vector<int> P;
};
```

See the CS 2704 notes on C++ Class Basics for more details and examples.

First, some differences versus Java:

- C++ classes are not specified with any access control on the class itself.
- C++ class declarations are statements, and hence are terminated with a semicolon.
- C++ class member functions are not usually implemented within the class declaration; rather the declaration is placed in a header file and the corresponding implementation is placed in a `.cpp` file.
- Members of a C++ class are `private` unless explicitly specified otherwise.

```
class Polynomial {
public:
    Polynomial();
    Polynomial(const string& N, const vector<int>& C);
    . . .
private:
    string Name;
    vector<int> P;
};
```

Implementing Member Functions

C++ Classes 3

Since the member function implementation is not incorporated into the class declaration, we need some extra syntax to tie the implementation to the class:

```
Polynomial::Polynomial(const string& N, const vector<int>& C) {  
    . . .  
}  
  
string Polynomial::getName() const {  
    . . .  
}
```

const Member Functions

C++ Classes 4

Member functions that do not modify any data members of the class can (and should) be declared with the modifier `const`.

`const` member functions:

- are prevented at compile-time from modifying any data members of the class
- may be invoked even on objects that are themselves declared as `const`

Accessor functions are typically `const`.

So are member functions and operators that create new objects from old ones.

```
class Polynomial {  
public:  
    . . .  
    Polynomial Derivative() const;  
    Polynomial Antiderivative() const;  
    int Integral(int a, int b) const;  
    . . .  
    string getName() const;  
};
```

C++ Operators

C++ Classes 5

In C++, operators may be overloaded, extending the language to apply familiar syntax to new types with natural semantics.

```
class Polynomial {
public:
    . . .
    Polynomial operator+(const Polynomial& RHS) const;
    Polynomial operator-(const Polynomial& RHS) const;
    Polynomial operator*(const Polynomial& RHS) const;
    int operator()(int x) const;
    . . .
};
```

Assuming that P, Q and R are objects of type Polynomial, all of the following are valid and have precisely the meaning you would expect:

```
P = Q + R;
P = Q * 5;
int Result = P(42);
```

We will explore operator overloading in great depth, but that will come a bit later.

Destructors

C++ Classes 6

C++ classes may include a special member function, whose name is the same as the class, preceded by a tilde character ('~').

This member is called the class *destructor*.

Whenever the lifetime of an object comes to an end, its destructor is automatically invoked. Destructors are rarely invoked explicitly.

The most common use for a destructor is to encapsulate the logic needed to deallocate any dynamic content for which an object is responsible.

Destructors may also be used for logging and other bookkeeping purposes.

```
class Polynomial {
public:
    . . .
    ~Polynomial();
    . . .
};
```

What's Automatic?

C++ Classes 7

C++ classes have certain features automatically.

If you do not provide any constructors, a default constructor will be created by the compiler; however it will be trivial (i.e., it will do nothing).

If you do not provide a destructor, a trivial one will be supplied by the compiler.

If you do not provide a copy constructor or assignment operator, simple non-trivial versions will be supplied by the compiler. More about these later...

Every C++ object has an implied data member named `this`, which is a pointer to the object itself.

Type Conversion Issues

C++ Classes 8

Any class constructor that takes a single value may be viewed as defining a conversion from objects of that type to objects of the class type.

For example:

```
class Foo {
public:
    Foo(int x);    // converts int to Foo
    . . .
};

Foo A(12);        // fine, just what we expect
A = 42;           // seemingly meaningless, right?
```

The last statement is, in fact, legal code. The `Foo` constructor is invoked automatically when the compiler attempts to make sense of the statement. This results in the creation of a new `Foo` object, which is then assigned to `A`.

This is probably not what we would want. If not, the automatic (or implicit) invocation of the constructor can be prevented:

```
class Foo {
public:
    explicit Foo(int x);
```

Copying Objects

C++ Classes 9

What happens when an object is copied?

By default, copying is performed either by a copy constructor or an assignment operator, depending on the context in which the copy is made.

The automatic copy constructor and assignment operator are very simple. They just copy the data members from the source object into the target object.

This is perfectly satisfactory for many classes:

```
class Complex {  
private:  
    double Real;  
    double Imaginary;  
    . . .  
};
```

```
Complex X(4.2, 7.3);  
Complex Y;  
  
Y = X;
```

This is called making a *shallow copy* because the logic does not extend beyond the level of the class data members.

Shallow Copy Problem

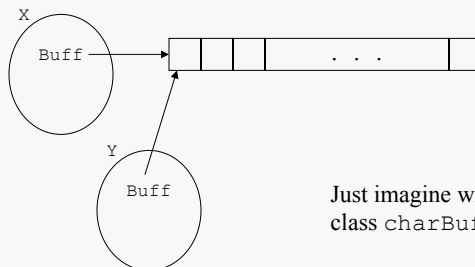
C++ Classes 10

But consider:

```
class charBuffer {  
public:  
    charBuffer(int Sz) {  
        Buff = new char[Sz];  
        . . .  
    }  
    . . .  
private:  
    char *Buff;  
    . . .  
};
```

```
charBuffer X(100);  
charBuffer Y;  
  
Y = X;
```

The effect is less than desirable:

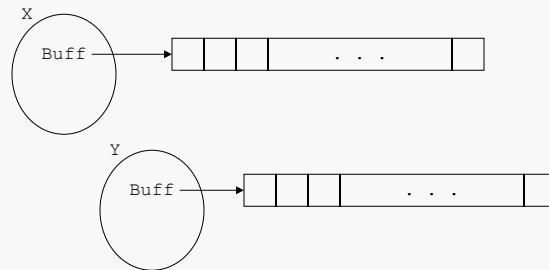


Just imagine what the destructor for the class `charBuffer` would do...

Deep Copy

C++ Classes 11

Clearly what we would really want would be for the assignment to have produced this situation:



And, of course, we would also want the contents of the array belonging to X to be duplicated in the array that belongs to Y.

This is called making a *deep copy*.

Since the automatic copy support for a C++ class does not make a deep copy, it must be the responsibility of the class to provide such support.

Implementing a Deep Copy Assignment

C++ Classes 12

For the previous example, what is needed is for the class `charBuffer` to include an implementation of the assignment operator that does, in fact, perform a deep copy:

```
charBuffer& charBuffer::operator=(const charBuffer& RHS) {  
  
    if ( this == &RHS ) // if self-assignment, no copying  
        return *this;  
  
    delete [] Buff;      // discard dynamic stuff in LHS object  
  
    Size = RHS.Size;    // copy static stuff into LHS object  
  
    Buff = new char[Size]; // make new array for LHS object  
  
    // copy array contents from RHS object into LHS object  
    for (unsigned int Pos = 0; Pos < Size; Pos++)  
        Buff[Pos] = RHS.Buff[Pos];  
  
    return *this;      // return RHS object (by reference)  
}
```

Deep Copy Constructor

C++ Classes 13

A *copy constructor* is just a constructor for a class X that takes an object of type X as its only parameter:

```
charBuffer::charBuffer(const charBuffer& Source) {  
  
    Size = Source.Size;    // copy static stuff into new object  
  
    Buff = new char[Size]; // make new array for new object  
  
    // copy array contents from source object into new object  
    for (unsigned int Pos = 0; Pos < Size; Pos++)  
        Buff[Pos] = Source.Buff[Pos];  
}
```

Note that the body of the copy constructor is a subset of the body of the assignment operator. This makes life simple.

Destructor Redux

C++ Classes 14

As noted before, the most common task for a class destructor is simply to deallocate anything an object of the class has allocated dynamically and retained control of:

```
charBuffer::~~charBuffer() {  
  
    delete [] Buff;  
}
```

Since this is invoked automatically whenever the lifetime of a `charBuffer` object comes to an end, and this does deallocate everything such an object could possibly have obtained dynamically, this would seem to guarantee that `charBuffer` objects will not cause memory leaks.

In this matter, things really are as they seem.

Automatic Cascading Effects

C++ Classes 15

When a shallow copy of an object is made, any data members that are objects are copied using the copy logic provided by their classes.

That's why the `Polynomial` class did not need its own copy constructor or assignment operator.

When an object is destructed, any data members that are objects have their own destructors invoked immediately after the destructor for the containing object.

That's why the `Polynomial` class did not need a user-defined destructor.

When an object is constructed, by default any data members that are objects have their own default constructors invoked immediately before the constructor for the containing object is executed.

Default Arguments

C++ Classes 16

When declaring a C++ function, you may optionally specify a default value for function parameters by listing initializations for them in the declaration:

```
class Polynomial {
public:
    Polynomial(const string& N = "no name",
               const vector<int>& C = vector<int>());
    . . .
};
```

You may specify defaults for some parameters and not specify defaults for others; however, all parameters w/o defaults must precede those that have defaults.

The default value is used if the call does not specify a corresponding actual parameter:

```
Polynomial F("F", vectorOfCoefficients);
Polynomial G("G");
Polynomial H;
```

Note that the modified constructor now plays the role of a default constructor.

Implicit Inline Functions

C++ Classes 17

C++ class member functions may also be implemented within the class declaration:

```
class Polynomial {
public:
    . . .
    string getName() const { // inline function
        return Name;
    }
    . . .
};
```

This may be considered bad practice since it exposes the details of the implementation to the user of the class.

However, this has an interesting side effect. Functions implemented within the class declaration are said to be *inline*.

The C++ compiler may optimize the compiled code by replacing calls to an inline function with the (suitably modified) function body, eliminating some work on the runtime stack.

Explicit Inline Functions

C++ Classes 18

An alternative is to specify the function is to be inlined by adding syntax to the function implementation:

```
// Polynomial.cpp
#include "Polynomial.h"

. . .
inline string Polynomial::getName() const {
    return Name;
}
. . .
```

This has the same effect, but does not expose the implementation in the header file.

Note: in any case, the compiler is free to ignore the request and NOT replace calls to the function with its body.