Classes in C++

Ajit Rajwade
Schaum Series Book: Material taken from chapters 8, 9, part of 13

Class in C++

- A structure is a compound datatype containing different members, each of which represents a feature of an object.
- Often, very specific operations/function are associated with an object.
- A **class** is a generalization of a structure which not only contains members (data) but also functions or operations that process the values of these members.
- A class is considered the foundation step in the so called **Object Oriented Programming Paradigm** or OOP.
- Note: in modern C++ editions, structures also contain member functions, though in the class C-language definition of structures, this provision was absent.

Class Declaration

- We will consider the example of a class for rational numbers.
- A rational number is essentially the ratio of an integer-valued numerator and denominator.
- Hence numerator and denominator are members of the class.
- Various operations can be performed on such rational numbers: printing the number, adding two numbers, reciprocal, conversion to decimals, and so on.
- The declaration for this class is given on the next slide.

```
class Rational {
public:
void assign(int, int);
double convert();
void invert();
void print();
private: int num, den;
};
```

```
int main(){
Rational x;
x.assign(22,7);
cout << "x = "; x.print();
cout << " = " << x.convert(); cout <<
endl;
x.invert();
cout << "1/x = "; x.print(); cout << endl;
}
```

- The class declaration is preferably outside any function. The class variables are declared inside a function or as global variables in the form: `Rational x;` (We say that `x` is an instance of class `Rational`)
- The numerator `num` and denominator `den` are the data members of the class. They are typically marked as `private`.
- The member functions `assign`, `convert`, `invert`, `print` are marked as `public`.
- Public members can be accessed from outside the class, whereas private members can be accessed only from within the class.
- This method of preventing access to data members from outside the class is called **information hiding**. It makes large softwares easier to maintain, understand and debug.
- To make it clear, private variables can be altered only within the member functions of a class. They cannot be accessed from outside. Hence statements such as `x.num = 3;` or `int y = x.num + x.den;` will produce syntax errors if they are included outside member functions.

```
void Rational:: assign(int numerator, int
denominator)
{
     num = numerator;
     den = denominator;
}

double Rational::convert()
{
     return double(num)/den;
}

void Rational::invert()
{
     int temp = num;
     num = den;
     den = temp;
}

void Rational::print()
{
     cout << num << " " << den;
}
```

OUTPUT:
x = 22 7 = 3.14286
l/x = 7 22

If the member functions are defined outside the body of the class, then a **scope resolution operator** :: is essential, which tells the compiler that these functions are intended to be member functions of the class `Rational`. Without this scope resolution operator, the compiler would have considered these to be any functions outside the class.

The function definitions **can also be written inside the class**. See the syntax on the next slide.

```
class Rational {
public:
void assign(int n, int d) { num = n; den = d; }
double convert() { return double(num)/den; }
void invert() { int temp = num; num = den; den = temp; }
void print() { cout << num << "/" << den; }
private:
int num, den;
};
```

Constructors

- The member function assign(int, int) assigns the values of numerator and denominator to an instance of the class.
- But it is elegant to be able to initialize these values right when the instance of the class Rational is **created**.
- This in the same spirit as initializations of the form: `int z = 50; char *s = “IITB”;`
- A way to do this is a C++ construct called a **constructor**.
- A constructor is a **member** function that is **automatically** called when a class instance is created or declared.
- The constructor must have the **same** name as the class.
- For example, see the code on the next slide.

```
class Rational {
public:
Rational(int n, int d) { num = n; den = d; }
void print() { cout << num << "/" << den; }
private:
int num, den;
};

int main()
{
Rational x(-1,3), y(22,7);
cout << "x = ";
x.print();
cout << " and y = ";
y.print();
}
```

OUTPUT:
x = -1/3 and y = 22/7

Constructors

- The effect of the constructor is exactly the same as the `assign(int,int)` member function.
- But the constructor is more elegant.
- When an object `x` is declared via the statement `Rational x(-1,3);`, the constructor is automatically called and it assigns the values -1 and 3 to `x.num` and `x.den`, the member variables of `x` via the parameters `n` and `d`.
- Thus the statement `Rational x(-1,3);` is equivalent to the following lines of code:

```
Rational x;

x.assign(-1,3);
```

Constructors

- A class may have multiple constructors, all of which have the same name, but have different parameter lists.
- This is an example of **function overloading** that we have seen earlier.
- For example, consider three different constructors for the class Rational here below:

```
class Rational {
public:
Rational() { num = 0; den = 1; }
Rational(int n) { num = n; den = 1; }
Rational(int n, int d) { num = n; den = d; }
void print() { cout << num << "/" << den; }
private:
int num, den;
};
int main(){
Rational x, y(4), z(22,7);
cout << "x = " ; x.print(); cout << endl;
cout << "y = " ; y.print(); cout << endl;
cout << "z = "; z.print(); cout << endl;
}
```

This version of the `Rational` class has three constructors.

The first has **no parameters** and initializes the declared object with the default values 0 and 1. This is the simplest one and is called the **default constructor**. if not expressly declared, it is created automatically for the class (as in the previous example)

The second constructor has one integer parameter and initializes the object to be the fractional equivalent to that integer.

The third constructor is the same as what we saw before.

Private member functions

- So far, we have seen private data members and public function members.
- However, some functions can be made private as well.
- Functions which accomplish very specific internal tasks, which an external user need not have knowledge of, are to be considered private functions.
- For example, when an instance of the class `Rational` is created, one may choose to divide both the numerator and denominator by their gcd to produce a “simpler” fraction.
- Such a detail need not be made available to the user, and hence a function to simplify such a fraction can be made private.
- See code on the next slide.

```
class Rational
{
    public:
    Rational(int n, int d) { num = n; den = d;  reduce(); }
    void print() { cout << num << "  "; cout << den << endl;}

    private:
    int num, den;
    int gcd(int j, int k) { if (j%k==0) return k; return gcd(k,j%k); }
    void reduce() { int g = gcd (num, den); num /= g; den /= g;}
};

int main()
{
    Rational x(100,360);
    x.print();
}
```

This class has two private functions: `void reduce()` and `int gcd(int, int)`. One could have included the entire `reduce()` function inside the constructor, but there is a basic principle in software: separate tasks should be handled by separate functions. A private member function can be called only from within the body of member functions of the class, not from outside the class. For example, x.reduce(); inside main will produce a syntax error in this example.

Copy Constructor

- Every class in C++ has two constructors created automatically.
- The first one is a default constructor that is invoked when an instance of the class is created. For example, the constructor `Rational()` in the previous slides.
- There is another constructor which copies the contents of one pre-defined instance of a class into another instance of the same class.
- This is called the **copy constructor**.
- This will be declared as follows: `Rational(const Rational &r) {num=r.num; den=r.den;}`
- Note the syntax of the copy constructor: it must have exactly one parameter which must be of the same type as the class being declared, and the parameter must be passed by reference.
- The complete code with a copy constructor is on the next slide.

```
class Rational
{
public:
        Rational(int n, int d) { num =n; den = d;reduce(); }

        Rational(const Rational& r) {num = r.num; den = r.den; }

        void print() { cout << num << "/" << den; }

private:
        int num, den;
        int gcd(int m, int n) { if (m%n==0) return n; return gcd(n,m%n);
}
        void reduce() { int g = gcd(num, den); num /= g; den /= g; }
};

int main()
{
        Rational x(100,360);
        Rational y(x);
        cout << "x = "; x.print();
        cout << ", y = "; y.print();
}
```

Copy constructors

- The copy constructor copies the `num` and `den` fields of the parameter `r` into the object being constructed. When `y` is declared, it calls the copy constructor which copies `x` into `y`.

- The copy constructor is called automatically whenever
  - an object is copied by means of a declaration initialization;
  - an object is passed by value to a function
  - When an object is returned by a function

- See example on the next slide.

```
class Rational{
public:
        Rational () { num = 0; den = 1;}
        Rational(int n, int d) { num =n; den = d;reduce(); }
        Rational(const Rational& r) {num = r.num; den = r.den; cout << "copy constructor"
<< endl;}

        void print() { cout << num << "/" << den; }

private:
        int num, den;
        int gcd(int m, int n) { if (m%n==0) return n; return gcd(n,m%n); }
        void reduce() { int g = gcd(num, den); num /= g; den /= g; }
};
Rational f(Rational r){ // copy constructor is invoked
        Rational s = r;
        return s;}

int main() {
        Rational x(22,7);
        Rational y(x); // copy constructor will be invoked
        Rational z;
        z = f(y); // copy constructor is invoked
}
```

Destructor

- When an object is created, a constructor is invoked.
- When a program moves out of the scope of an object, the object's "life comes to an end".
- A default member function called the **destructor** is invoked in such a case.
- Each class has exactly one destructor, even though it may have multiple constructors.
- The programmer may write a special destructor, if not a default destructor is always invoked.
- The destructor always has the form: `~<classname>() { // code here }`

```
class Rational {
public:
Rational() { cout << "OBJECT IS BORN.\n"; }
~Rational() { cout << "OBJECT DIES.\n"; }
private:
int num, den;
};

int main()
{
        {
                Rational x; // beginning of scope for x
                cout << "Now x is alive.\n";
        } // end of scope for x
        cout << "Now between blocks.\n";
        {
                Rational y;
                cout << "Now y is alive\n";
        }// end of scope for x
}
```

Output:
OBJECT IS BORN.
Now x is alive.
OBJECT DIES.
Now between blocks.
OBJECT IS BORN.
Now y is alive
OBJECT DIES.

Although the system will provide them automatically, it is considered good programming practice always to define the copy constructor, the assignment operator, and the destructor within each class definition.

Difference between `struct` and `class`

- In a class, all members are `private` by default unless declared to be `public`.
- In a struct, all members are `public` by default unless declared to be `private`.
- A `struct` in C++ too can contain member functions - I did not mention this fact to you when we did the lectures on structures.
- Structures in C did not have the provision of member functions - they contained only data members.
- Things such as pointers to structures, structure of pointers, structures as parameters to a function, dynamic memory allocation of structures, etc. are **also applicable to classes**!

Static data members

- In some cases, a **single** value of a data member is applicable to all instances of that class.
- In such a case, there is no use to store the data member in every instance.
- A provision for such an entity exists via **static** data members.
- A static data member is declared simply by including the word `static` before the variable's data type.
- A static member may be public or private.
- A good example of a static member is one that counts the **number of instances** of a class.

```
class Widget {
public:
// increment counter when instance is created
Widget() { ++count; }
// decrement counter when instance is created
~Widget() { --count; }
static int count;
};
int Widget::count = 0; // definition of count (must be outside all
functions)
int main()
{
        Widget w, x;
        cout << "Now there are " << w.count << " widgets.\n";
        {
                Widget w, x, y, z;
                cout << "Now there are " << w.count << "widgets.\n";
        }
        cout << "Now there are " << w.count << " widgets.\n";
        Widget y;
        cout << "Now there are " << w.count << " widgets.\n";
}
```

OUTPUT:
Now there are 2 widgets.
Now there are 6widgets.
Now there are 2 widgets.
Now there are 3 widgets.

We could access `w.count` from outside the class as it was public.
A static member is like a global variable. It is shared across all instances of the object.
However a static member may even be private: see example on next slide.

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

class Widget {
public:
Widget() { ++count; }
~Widget() { --count; }
int numWidgets() { return count; }
private:
static int count;
};

int Widget::count = 0;

int main ()
{
Widget w, x;
cout << "Now there are " << w.numWidgets() << " widgets.\n";
{
Widget w, x, y, z;
cout << "Now there are " << w.numWidgets() << " widgets.\n";
}
cout << "Now there are " << w.numWidgets() << " widgets.\n";
Widget y;
cout << "Now there are " << w.numWidgets() << " widgets.\n";
}
```

OUTPUT:
Now there are 2 widgets.
Now there are 6 widgets.
Now there are 2 widgets.
Now there are 3 widgets.

As `count` is now private, we can access it only via a member function, in this case `numWidgets()`.

Static Member Functions

- In the previous slide, we are always calling `numWidgets()` via the instance `w` to get the count.
- Since `count` is a static data member, it doesn't matter which instance is used here, as long as the instance is in scope. Note that `w` was always in scope at every place where this function was invoked, unlike some other instances (`y`, `z` for example)
- However this is inelegant. Why should you have to refer to any instance at all to get the count?
- Moreover, the function `numWidgets()` cannot be called without at least one instance of the object being created.
- A way out of this is the usage of a **static member function** to access all static variables.
- A static member function is independent of any specific instance of the class.
- It is simply invoked via a scope resolution operator, like you will see on the next slide.

```
class Widget{
public:
Widget() { ++count; }
~Widget() { --count; }
static int num() { return count; }

private:
static int count;
};

int Widget::count = 0;
int main()
{
        cout << "Now there are " << Widget::num() << " widgets\n";
        Widget w, x;
        cout << "Now there are " << Widget::num() << " widgets\n";

        {
                Widget w, x, y, z;
                cout << "Now there are " << Widget::num() << " widgets\n";
        }
        cout << "Now there are " << Widget::num() << " widgets\n";

        Widget y;
        cout << "Now there are " << Widget::num() << " widgets\n";
}
```

```
Now there are 0 widgets
Now there are 2 widgets
Now there are 6 widgets
Now there are 2 widgets
Now there are 3 widgets
```

Static Member Functions

- We are declaring the `num()` function to be static in order to make it not dependent on any specific instance of the class.
- The function is invoked as a member of the `Widget` class by using the scope resolution operator `::`, i.e. as `Widget::num()`.
- As the function was declared `static`, it can be called before any instances of this class have been created.

A String class

- We will implement a C++ class to create and process a character string.
- The data members will be a character array (the actual string) and a string length.
- A constructor with a length parameter will be used to dynamically allocate memory for the string and fill it up with space (‘ ‘) characters followed by a NULL.
- The destructor will deallocate the memory allocated for the string.
- There will be a copy constructor which will copy the data from one instance to another.
- There will be other member functions to print the string, determine its length, determine the character at a certain location, convert the string into a character array and so on.

```
class MyString{
public: MyString(int); // default constructor

MyString(const char*); // constructor

MyString(const MyString&); // copy constructor

~MyString() { delete [] data; } // destructor

int length() { return len; } // access function
char* convert() { return data; } // access function
char character(short i) { char c = data[i]; return c; }
void print() { cout << data; }

private:
int len; // number of (non-null) characters in string
char* data; // the string
};
```

```
MyString::MyString(int size) {
    len = size;
    data = new char[len+1];
    for (int i=0; i < len; i++) data[i] = ' ';
    data[len] = '\0';
}

MyString::MyString(const char* str) {
len = strlen(str);
data = new char[len+1];
strcpy(data, str);
}

MyString::MyString(const MyString& str) {
    len = str.len;
    data = new char[len+1];
    strcpy(data, str.data);
}
```

```
int main(){
        MyString t("IITBombay_CS101");
        MyString u(t);
        int i = 3;

        cout << "The string t is " ; t.print(); cout << endl;
        cout << "The string u is " ; u.print(); cout << endl;
        cout << "Length of t = " << t.length() << endl;
        cout << "Length of u = " << u.length() << endl;
        cout << "The character at location " << i << " in u is "
<< u.character(i) << endl;
        cout << "The character at location " << i+1 << " in t is
" << u.character(i+1) << endl;
}
```

The string t is IITBombay_CS101
The string u is IITBombay_CS101
Length of t = 15
Length of u = 15
The character at location 3 in u is B
The character at location 4 in t is o

Operator Overloading

- With basic datatypes such as `int`, `double`, `char`, etc, the assignment operator is used frequently.
- Can it be used for instances of a class?
- In the class `Rational`, we have seen different types of constructors.
- We have also seen the `assign()` member function.
- But can we write statements such as `x = y = z;` where x,y,z are instances of class `Rational`?
- Yes, via a construction called operator overloading, where the `=` operator for the `Rational` class is defined, in a manner similar to a member function.

```
class Rational{
public:
Rational(int x, int y){ // default constructor
        num = x; den = y;
}
Rational(const Rational& r){ // copy constructor
        num = r.num;
        den = r.den;
}
void operator=(const Rational& r){ // assignment operator
        num = r.num;
        den = r.den;
}
void print() {
        cout << endl << num << " " << den;
}
// other declarations go here
private:
int num;
int den;
};
int main() {
        Rational x(1,10), y(2,7);
        x = y; // the contents of y will be copied into x
        x.print();
        y.print();
}
```

Note in the statement `x = y`, it is `x` which owns this call. This is a member function of x, and y is a parameter passed by reference.

Multiple Assignments and the `this` operator

- Consider C++ statements like `x = y = z = 3.2` where `x`, `y`, `z` are doubles.
- The value of 3.2 gets assigned to `z`, and this same value then gets assigned to `y`, and finally to `x`.
- The = operator can act like a function (`f`) similar to the one defined in the previous slide.
- In the chain `x = y = z = 3.2`, the function `f` is called three times.
- The function `f` should return a double. The **nested** calls are of the form `f(x,f(y,f(z,3.2))`.
- The assignment operator is thus a function that should **return** the value that it assigns.
- So we can express it in the following form: `Rational& operator = (const Rational& r)` which is defined in the next slide. This allows for **chaining** of assignments in the form `x = y = z;`
- The function should return a **reference** to the same type as the object being assigned.
- When the = operator is being overloaded as a member function, it should return a **reference** to the same object of which it is a member.
- This is accomplished via the `this` operator (there is no name available for the object otherwise inside the body of the class).

```
class Rational{
public:
Rational(int x, int y){ // default constructor
        num = x; den = y;
}
Rational(const Rational& r){ // copy constructor
        num = r.num;
        den = r.den;
}
Rational& operator=(const Rational& r){ // assignment operator
        num = r.num;
        den = r.den;
        return *this;
}
void print() {
        cout << endl << num << " " << den;
}
// other declarations go here
private:
int num;
int den;
};
int main() {
        Rational x(1,10), y(2,7), z(1,6);
        x = y = z; // the contents of y will be copied into x
        x.print();
        y.print();
        z.print();
}
```

Overloading arithmetic operators

- Suppose you wanted to add one rational number to another.
- That could be accomplished via a member function as follows:

```
void Rational::add(Rational &r){

num = num*r.den+ den*r.num;

den = den*r.den;

}
```

- This can also be accomplished by overloading the += operator:

```
void Rational::operator += (Rational &r){

num = num*r.den+ den*r.num;

den = den*r.den;

}
```

- The full code for this is on the next slide.

```
class Rational{
public:
Rational(int x, int y){ // default constructor
        num = x; den = y;
}
Rational(const Rational& r){ // copy constructor
        num = r.num;
        den = r.den;
}
Rational& operator=(const Rational& r){ // assignment operator
        num = r.num;
        den = r.den;
        return *this;
}
void print(){ cout << endl << num << " " << den;}
void add(Rational &r){
num = num*r.den + den*r.num;
den = den*r.den;
}
void operator +=(Rational &r){
num = num*r.den + den*r.num;
den = den*r.den;
}
// other declarations go here
private:
int num;
int den;
};
```

```
int main(){
Rational x(1,10), y(2,7),
z(1,6);
x.add(y);
x.print();
y += z;
y.print();
}
```

Overloading arithmetic operators

- Can we do something like `x = y+z;` where `x,y,z` are objects of class `Rational`?
- Yes, by overloading the `+` operator inside the `Rational` class and returning an object of type `Rational` from within the body of the defined operator.
- For example, by including the following inside `Rational`:

```
Rational Rational::operator+(const Rational& r){

Rational z;

z.num = num*r.den+den*r.num; z.den = den*r.den;

return z;

}
```

- See next slide for full code.

```
class Rational{
public:
Rational () { num =0; den = 1;}
Rational(int x, int y){ // default constructor
        num = x; den = y;
}
Rational(const Rational& r){ // copy constructor
        num = r.num;
        den = r.den;
}
Rational& operator=(const Rational& r){ // assignment
operator
        num = r.num;
        den = r.den;
        return *this;
}
Rational operator+(const Rational& r){
Rational z;
z.num = num*r.den+den*r.num; z.den = den*r.den;
return z;
}
void print(){ cout << endl << num << " " << den;}
// other declarations go here
private:
int num;
int den;
};
```

```
int main() {
Rational x(1,2), y(1,2), z(1,4);

x.add(y);
x.print();
y += z;
y.print();

x = y+z;
x.print();
}
```

What do you think will happen in this code if the first constructor Rational() were excluded?

Overloading arithmetic and **comparison** operators

- We overloaded the + operator.
- Of course, very similarly, we can overload the operators -, *, / and so on.
- Try those out yourself!
- You can overload comparison operators like ==, >=, > , <=, < as well.
- For example, for ==:

```
bool Rational::operator==(const Rational& r){

return (num*r.den == den*r.num);

}
```

- Try out the >=, >, <, <= operators as well.

Overloading unary operators

- Let us overload the ! operator to implement reciprocals for the Rational class.

```
Rational& Rational::operator!(){
     int temp = num;
     num = den;
     den = temp;
     return *this;
}
```

- Let us overload the ++ (increment) operator for the Rational class. Here, we will not be able to distinguish between pre- and post-increment.

```
Rational& Rational:: operator++(){
     num += den;
     return *this;
}
```

```
class Rational{
public:
Rational () { num =0; den = 1;}
Rational(int x, int y){ // default constructor
        num = x; den = y;
}
Rational(const Rational& r){ // copy constructor
        num = r.num;
        den = r.den;
}
Rational& operator=(const Rational& r){ // assignment
operator
        num = r.num;
        den = r.den;
        return *this;
}
Rational operator+(const Rational& r){
Rational z;
z.num = num*r.den+den*r.num; z.den = den*r.den;
return z;
}
void print(){ cout << endl << num << " " << den;}

Rational& operator!() {int temp = num; num = den; den = temp;
return *this;}

Rational& operator++() {num += den; return *this;}
```

```
// other declarations go here
private:
int num;
int den;
};
```

```
int main() {
Rational x(5,7);
x =!x;
x.print();
x=++x;
x.print();
}
```

Overloading the subscript operator

- The array subscript operator is used to access the i-th element of an array.
- We will overload the subscript operator to access data members of the Rational class - index of 0 returns the numerator and index of 1 returns the denominator.

```
int operator[](int i){

if (i == 0) return num; else return den;

}
```

- In the main function, the relevant syntax is: `Rational x(1,2); int z1 = x[0]; int z2 = x[1];`
- This example is somewhat contrived, but this can instead be used in the `MyString` class.

Class for a Matrix

- We will implement a class called `MyMatrix` to implement a 2D array of numbers (matrix).
- **Private data members:** number of rows, number of columns, 2D data array containing the actual matrix entries
- (public) **Constructor:** performs dynamic memory allocation given a certain number of rows and columns. By default, all data values are set to 0.
- (public) A **copy constructor** to copy all data values from an object
- (public) **Destructor:** deletes the 2D array
- Public routines to print the values of a matrix, and to input a matrix from the user

Class for a Matrix

- In addition, we will overload operators to perform matrix assignment and addition.
- The assignment operator is declared as follows: `MyMatrix& operator=(const MyMatrix& r);`
- The addition operator is declared as follows: `MyMatrix operator+(const MyMatrix& z);`
- We will also overload the += operator whose declaration is: `void operator+=(const MyMatrix& z);`
- See next few slides for the full code.

```
class MyMatrix
{
public:
     MyMatrix (int n1, int n2);
     MyMatrix (const MyMatrix&); //copy constructor
     void inputMatrix ();
     void printMatrix ();
     ~MyMatrix();
     MyMatrix& operator=(const MyMatrix& r);
     MyMatrix operator+(const MyMatrix& z);
     void operator+=(const MyMatrix& r);

private:
     int nc, nr, **data;
     int **allocate_2d (int n1, int n2); // orivate fn. For dyn. Mem. alloc. Of a 2d
array
     void deallocate_2d (int **A, int n1, int n2); // priv. Fn. for deallocation
};
```

```
MyMatrix::MyMatrix (int n1, int n2){ //constructor
            nr = n1;
            nc = n2;
            data = allocate_2d(nr,nc);
            for(int i=0;i<nr;i++){
                    for(int j=0;j<nc;j++){
                            data[i][j] = 0;
                    }
            }
    }

    MyMatrix::MyMatrix (const MyMatrix& r) { // copy constructor
            cout << endl << "CC";
            nr = r.nr; nc = r.nc;
            data = allocate_2d(nr,nc);
            for(int i=0;i<nr;i++){                  {
                    for(int j=0;j<nc;j++){
                            data[i][j] = r.data[i][j];
                    }
            }
    }
```

```
void MyMatrix::inputMatrix (){ // member function for keyboard input
for(int i=0;i<nr;i++){
for(int j=0;j<nc;j++){
cout << "enter value at index " << i << "," << j << ": ";
cin >> data[i][j];
}
}
}

void MyMatrix::printMatrix () { // member function for output on screen
for(int i=0;i<nr;i++){
for(int j=0;j<nc;j++){
cout << data[i][j] << " ";
}
cout << endl;
}
}

MyMatrix::~MyMatrix(){ // destructor
cout << "DD";
deallocate_2d(data,nr,nc);
}
```

```
MyMatrix& MyMatrix::operator=(const MyMatrix& r){ // assignment operator
 nc = r.nc; nr = r.nr;
 cout << "AE";
 for(int i=0;i<nr;i++){
 for(int j=0;j<nc;j++){
 data[i][j] = r.data[i][j];
 }
 }
 return *this;
}

MyMatrix MyMatrix::operator+(const MyMatrix& z){
if (nr != z.nr || nc != z.nc) { cout << "Incompatible sizes" ; }
MyMatrix s(nr,nc);
for(int i=0;i<nr;i++){
for(int j=0;j<nc;j++){
s.data[i][j] = data[i][j] + z.data[i][j];
}
}
return s;
}
```

```
// assignment operator
void MyMatrix::operator+=(const MyMatrix& r){
nc = r.nc;
nr = r.nr;
for(int i=0;i<nr;i++){
for(int j=0;j<nc;j++){
data[i][j] += r.data[i][j];
}
}
}
int **MyMatrix::allocate_2d (int n1, int n2){
int **A = new int*[n1]; // array of n1 int* variables
for(int i=0;i<n1;i++)
{A[i] =  new int [n2];} // array of n2 int variables
return A;
}
void MyMatrix::deallocate_2d (int **A, int n1, int n2){
for(int i=0;i<n1;i++)
{delete [] A[i];} // array of n2 int variables
delete[] A;
}
```

```
int main()
{
        int nr, nc;
        cout << "enter the number
of rows and columns: ";
        cin >> nr >> nc;

        MyMatrix m1(nr,nc);
        m1.inputMatrix();
        m1.printMatrix();


        MyMatrix m2(nr,nc);
        m2.inputMatrix();
        m2.printMatrix();


        MyMatrix m3(nr,nc);
        m3 = m1 + m2;
        m3.printMatrix();
}
```

Class MyMatrix

- What happens if you run the code without including the overloading of the assignment operator?
- There will be no compilation error, as there is a default assignment operator.
- But there will be run time errors such as a segmentation fault - try it out. This is because the result of the addition m1+m2 will not get copied into m3 properly if the operator = is not overloaded.
- Detailed reason: See next slide.
- Notice the usage of private functions for memory allocation!
- Why is the destructor invoked 4 times in the above code though there are only three `MyMatrix` variables created in `main`? Because at the end of the + operator, the local object s is destroyed. That gives rise to the fourth invocation of the destructor.

Class MyMatrix

- Inside `+`, the local object s is created. In the statement `m3 = m1+m2`, the object s is created.
- The default = operator in C++ will copy `s.nr` and `s.nc` into `m3.nr` and `m3.nc` respectively. It will also copy the **base address** of `s.data` into `m3.data`.
- When the function for `+` finishes, the local object `s` is destroyed and the memory associated with `s.data` is deleted. This happens due to the way the destructor has been written.
- Now, when you try to access `m3.data` from `m3.printMatrix()`, you are essentially trying to work with a **dangling pointer**.
- This causes a segmentation fault.
- Instead, the way we have overloaded the = operator, we copy the **entire array** (all `n1*n2` elements) from `s.data` into `m3.data`. The address of `m3.data` remains **different** from that of `s.data`.
- Hence even when `s` is destroyed, you have no problems accessing `m3.data`!

Class Templates

- We saw the implementation of a matrix (of integers) in the previous slide.
- What if you wanted to build a matrix of floats or doubles, or maybe of complex-valued numbers?
- One way is to write a separate class for integer matrix, float matrix, double matrix, etc.
- Instead of so much redundancy we may write a matrix class with a template parameter, i.e. a parameterized datatype.
- The syntax for this is on the next slide.

```
template <class T> // T is the parameterized datatype
class MyMatrix
{
public:
    MyMatrix (int n1, int n2);
    MyMatrix (const MyMatrix&);
    void inputMatrix ();
    void printMatrix ();
    ~MyMatrix();
    MyMatrix& operator=(const MyMatrix& r);
    MyMatrix operator+(const MyMatrix& z);
    void operator+=(const MyMatrix& r);

private:
    int nc, nr;
    T **data;
    T **allocate_2d (int n1, int n2);
    void deallocate_2d (T **A, int n1, int n2);
};
```

```
T** MyMatrix::allocate_2d (int n1, int n2){
    T** A = new T*[n1]; // array of n1 int* variables
    for(int i=0;i<n1;i++){
    A[i] =  new T [n2]; // array of n2 int variables
    }
    return A;
}

void MyMatrix::deallocate_2d ( T** A, int n1, int n2){
    for(int i=0;i<n1;i++){
    delete [] A[i]; // array of n2 int variables
    }
    delete[] A;
}
```

```
int main()
{
        int nr, nc;
        cout << "enter the number of rows and columns: ";
        cin >> nr >> nc;

        MyMatrix<double> m1(nr,nc);
        m1.inputMatrix();
        m1.printMatrix();


        MyMatrix<double> m2(nr,nc);
        m2.inputMatrix();
        m2.printMatrix();


        MyMatrix<double> m3(nr,nc);
        m3 = m1 + m2;
        m3.printMatrix();
}
```

General syntax for templated classes

- The basic syntax is:

```
template <class T, …> // list of template parameters and non-template parameters

class MyClass{ // class declaration

};
```

- The class `MyClass` may contain data members of type T in the example above.
- Note that the parameter list may contain one or more template parameters and one or more non-template parameters. For example:

```
template <class T, int n, …>

class MyClass {};
```

- Templates are instantiated at compile time. The values passed to non-template parameters must be constants.
- See example on next slide.

```
template<class T, int n>
class X {};

int main() {
X<float, 22> x1; // OK
const int n = 44;
X<char, n> x2; // OK
int m = 66;
X<short, m> x3; // ERROR: m must be constant
}
```

Stack: Data Structure

- A **stack** is a data structure used for storage of data.
- The stack involves an **array** of data (of any single datatype)and a special index called the “**top**” (or “top of the stack”).
- Let us call the array A.
- When the stack is **empty**, the top is set to -1. Likewise top being -1 indicates that the stack is empty.
- While adding an element x to the stack, the top is incremented by one and we perform the assignment A[top] = x.
- This is called **pushing** an element onto the stack.
- While the array A may be very large, the only valid indices are from 0 to the top (both inclusive).

https://en.wikipedia.org/wiki/Stack_(abstract_data_type)#/media/File:Tallrik_-_Ystad-2018.jpg

Implementation of a Stack using Templates

- For deleting an element from the stack (called **popping** off the stack), the value of top is decremented by 1.
- At that point, the array entries A[top+1] onwards are all invalid.
- If the size of A is n, then the stack is said to be **full** if top is equal to n-1.
- We can implement a stack using templates, but we will first do it with just integers.
- Note: both addition and deletion of elements of a stack are performed at the **top** of the stack.
- For this reason, a stack is said to be a **last in first out** (**LIFO**) data structure.

What is the use of a stack?

- A stack is used in every operating system for **nested function calls**.
- Consider that a function f1 calls another function f2, which calls function f4. Then once f4 and f2 are over, f1 calls function f3.
- Before passing control to f2, the activation frame of f1 is pushed onto the program stack.
- Then f2 executes. Before passing control to f4, the AF of f2 is pushed onto the stack.
- Once f4 finishes its execution, the AF of f2 is popped off the stack and control reaches f2.
- Once f2 finishes execution, the AF of f is popped off the stack and control reaches f1.
- When f1 reaches f4, the AF of f1 is again pushed onto the stack.
- Then f4 is called, and after f4 finishes, the AF of f1 is popped off the stack.

What is the use of a stack?

- You can use your own created stack to convert any recursive program to a non-recursive one.
- Essentially, the system stack used in the recursive calls is just replaced by your implementation.
- Thus, you could use the stack to cleverly write a non-recursive version of mergesort, quicksort, fractal rendering, determinant computation and many others!

MyStack Class

- Private members: pointer to data array, top (index) and size of array
- Public: Constructor to allocate memory for the array based on user-specified size
- Public: Destructor for memory deallocation
- Public: member functions to push, pop, check whether the stack is empty or full

```
class MyStack{
        private:
        int top;
        int size; // if a stack is implemented using dynamic
allocation, you should store the size
        int *A;

        public:
        bool isEmpty() {return (top==-1);}
        bool isFull() {return (top > -1 && top==size-1);}
        void push (int q){
                if (!isFull()) A[++top] = q;
                else cout << endl << "stack is full";
        }
        int pop (){
                if (!isEmpty()) {top--;  return A[top+1];}
                return INVALID;
        }
        int getTop() {return top;}
        MyStack() {top=-1; A = NULL; size = 0;}
        MyStack(int n) { top=-1; size = n; A = new int[size];}
        ~MyStack() { delete [] A; top=-1; size =0; A = NULL;}
};
```

```
int main(){
        int s,q;
        cout << "enter size of the stack";
        cin >> s;
        MyStack m1(s);

        while (true)        {
                cout << endl;
                cout << "Choose one of the following: (1) push, (2)
pop, (3) exit";
                int choice;
                cin >> choice;

                if (choice == 3) break;
                if (choice == 1){
                        cout << "enter the element to push:";
                        cin  >> q;  m1.push(q);
                        cout << " the top of the stack is " <<
m1.getTop();
                }
```

```
                if (choice == 2){
                        q = m1.pop();
                        if (q != INVALID)
                        {
                                cout << endl << "the element popped off is " <<
q;
                        }
                        else
                        {
                                cout << endl << "The stack is empty";
                        }
                        cout << " the top of the stack is " <<  m1.getTop();
                 }
        }
}
```

```
$ ./mystack.o
enter size of the stack4

Choose one of the following: (1) push, (2) pop, (3) exit1
enter the element to push:10
 the top of the stack is 0
Choose one of the following: (1) push, (2) pop, (3) exit1
enter the element to push:20
 the top of the stack is 1
Choose one of the following: (1) push, (2) pop, (3) exit1
enter the element to push:30
 the top of the stack is 2
Choose one of the following: (1) push, (2) pop, (3) exit1
enter the element to push:40
 the top of the stack is 3
Choose one of the following: (1) push, (2) pop, (3) exit1
enter the element to push:50

stack is full the top of the stack is 3
```

```
Choose one of the following: (1) push, (2) pop, (3) exit2

the element popped off is 40 the top of the stack is 2
Choose one of the following: (1) push, (2) pop, (3) exit2

the element popped off is 30 the top of the stack is 1
Choose one of the following: (1) push, (2) pop, (3) exit2

the element popped off is 20 the top of the stack is 0
Choose one of the following: (1) push, (2) pop, (3) exit2

the element popped off is 10 the top of the stack is -1
Choose one of the following: (1) push, (2) pop, (3) exit2

The stack is empty the top of the stack is -1
Choose one of the following: (1) push, (2) pop, (3) exit2

The stack is empty the top of the stack is -1
Choose one of the following: (1) push, (2) pop, (3) exit3
```

Stack with Templates

- We will use T as the placeholder for the datatype parameter.
- T could be used for an integer, double, char, bool, etc.
- It can also be used for classes such as MyString or Rational, or any other class you define - in that case, your .cpp program file must contain definitions of those classes as well.

```
template <class T>
class MyStack{
        private:
        int top;
        int size; // if a stack is implemented using dynamic allocation,
you should store the size
        T *A;
        public:
        bool isEmpty() {return (top==-1);}
        bool isFull() {return (top > -1 && top==size-1);}
        void push (T q){
                if (!isFull()) A[++top] = q;
                else cout << endl << "stack is full";
        }
        T pop (){
                if (!isEmpty()) {top--;  return A[top+1];}
                return INVALID;
        }
        int getTop() {return top;}
        MyStack() {top=-1; A = NULL; size = 0;}
        MyStack(int n) { top=-1; size = n;  A = new T[size];}
        ~MyStack() { delete [] A; top=-1; size =0; A = NULL;}
};
```

```
int main()
{
        int s;
        double q;
        cout << "enter size of the stack";
        cin >> s;

        MyStack<double>m1(s);


         // code just as before (in stack without templates)
}
```

Stack: A few comments

- A stack need not be implemented only using an array - there are implementations that use linked lists as well.
- There is a data structure called the queue where elements are added at one of the array (called rear) but elements are deleted from the other end (called front).
- A queue is a FIFO (first in first out) data structure where a stack is LIFO (last in, first out).