Nested Classes

- Like blocks and namespaces, classes are scopes and can nest.
- Nesting allows local hiding of names and local allocation of resources.
- This is often desirable when a class is needed as part of the implementation of a larger construct.

Example:
```cpp
char c;             // external scope ::c
class X            // outer class declaration X::
{
    public:
        char c;        // X::c
        class Y       // inner class declaration X::Y::
        {
            public:
                void foo(char e)  {  X t; ::c = t.X::c = c = e; }
            private:
                char c;   // X::Y::c
        };
};
```

Static and Const Members

- Using the modifier `static` in declaring a data member means that the data member is independent of any given class variable.
- The data member is part of the class but separate from any single class object.
- You remember that nonstatic data members are created for each instance of the class.
- Using static data allows class data to be scoped to the class but still require only one object for its storage.
- Without static data members, data required by all instances of a class would have to be global.

Example:
```cpp
class Point
{
public:
    static int how_many;
};
Point::how_many = 0;
...
++Point::how_many;
```

Static and Const Members

- A data member declared with the `const` modifier cannot be modified after initialization.
- syntactically, a const member function has the modifier follow the argument list inside the class declaration.

Example:
```cpp
class Foo
{
    static int foo_function();
};
int Foo::foo_function()
{
    ...  
}
```
static and const Members

• The const and static member function implementation can be understood terms of this pointer access.
• An ordinary member function is invoked as x.fcn(i, j, k).
• It has an explicit argument list i, j, k and an implicit argument list that includes the members of x (accessible through the this pointer).
• A static member function does not get the implicit arguments.
• A const member function cannot modify its implicit arguments.

The this Pointer

• The keyword this denotes a self-referential pointer to a class object.
• It cannot be used in static member functions.

Example:
class Point
{
public:
    void init(double u, double v) { x = u; y = v; }
    Point inverse() { x = -x; y = -y; return (*this); }
    Point* where_am_I() { return this; }
private:
    double x, y;
};

Constructors and Destructors

• A constructor is a member function whose name is the same as the class name.
• It constructs values of the class type.
• This process involves initializing data members and, frequently, allocating free store by using new.
• A destructor is a member function whose name is the class name preceded by the ~ character.
• It finalizes objects of the class type.
• Typically, a destructor deallocates store assigned to the object by using delete.

Constructors and Destructors

Constructors
• can take arguments,
• can be overloaded.

A constructor is invoked whenever
• its associated type is used in a definition,
• call-by-value is used to pass a value to a function,
• the return value of a function must create a value of associated type.

Classes with Constructors

Example: A data type ModInt for storing numbers that are computed with a modulus.
class ModInt
{
public:
    ModInt(int i);  // constructor declaration
    void assign(int i) { v = i % modulus; }
    void print() const { cout << v << ‘\n’; }
    const static int modulus;
private:
    int v;
};
ModInt::ModInt(int i) { v = i % modulus; }  // constructor definition
const int ModInt::modulus = 60;
Classes with Constructors

void main()
{
    ModInt a(5);
    ModInt b(62);
    a.print();
    b.print();
}

What does the output look like?
5
2

Classes with Constructors

What happens if we declare a variable c as follows:

ModInt c;

Since this class has only one constructor, and this constructor needs one int argument, this declaration causes a compile-time error.

The declaration above requires a default constructor.

The Default Constructor

• A constructor requiring no arguments is called the default constructor.

• It can be a constructor with an empty argument list or one whose arguments all have default values.

• It has the special purpose of initializing arrays of objects of its class.

In the ModInt example, it would be useful to define a default value of v to be 0.

To achieve this, we could add the following default constructor:

ModInt() { v = 0; }

The Default Constructor

The Default Constructor

main ()
{
    ModInt s1, s2;
    ModInt d[5];
    ModInt s1.print();
    ModInt s2.print();
    ModInt d[3].print();
}

Output:
0
0
0

Constructor Initializers

• A special syntax is used for initializing class members.

• Constructor initializers for class members can be specified in a comma-separated list that follows the constructor parameter list.

• The previous example can be recoded as:

ModInt(int i = 0): v(i % modulus) {}

• Notice that initialization replaces assignment.

• The individual members must be initializable as member-name (expression list).
Constructors as Conversions

- Constructors of a single parameter are used automatically for conversion unless declared with the keyword explicit.
- For example, T1::T1(T2) provides code that can be used to convert a T2 object to a T1 object.
- Let us take a look at the following class PrintChar, whose purpose is to print invisible characters with their ASCII designation (for example, the code 07 is alarm or bel).

```cpp
class PrintChar
{
public:
    PrintChar(int i = 0) : c(i % 128) {}
    void print() const { cout << rep[c]; }
private:
    int c;
    static const char* rep[128];
};
const char *PrintChar::rep[128] = {"nul", "soh", "stx", ..., "}", ";", "del"};
```

Constructors as Conversions

```cpp
int main()
{
    PrintChar c;
    for (int i = 0; i < 128; i++)
    {
        c = i;    // or: c = static_cast<PrintChar>(i);
        c.print();
        cout << endl;
    }
}
```

This program prints out the first 128 ASCII characters or their printable representations.

Structure Pointer Operator

- For accessing members in structures and classes we have used the member operator ‘.’.
- Now we introduce the structure pointer operator ‘->’, which provides access to the members of a structure via a pointer.
- If a pointer variable is assigned the address of a structure or class object, a member of the object can be accessed by the ‘->’ operator.

```cpp
Example:
int main()
{
    MyClass a, *b;
    a.SetValue(9);
    b->SetValue(5);
    (*b).SetValue(7);
}
```

Member access is correct in all three cases (but note that we should allocate memory for b first).

Constructors and Destructors

Now we are going to talk about

- two examples for classes with constructors and destructors:
  - a singly linked list
  - a two-dimensional array
- overloaded operators and user-defined conversions
A Singly Linked List

• We are going to develop a **singly linked list datatype**.
• This is the prototype of many useful dynamic abstract data types (ADTs) called **self-referential structures**.
• Self-referential structures have pointer members that refer to objects of their own type.
• Such structures are the basis of many useful container classes.

A Singly Linked List

We want to be able to perform the following list operations:

• **Prepend**: adds an element to the front of the list
• **First**: returns the first element in the list
• **Print**: prints the list contents
• **Del**: deletes the first element in the list
• **Release**: destroys the list

```cpp
struct SListElem
{
    char data;
    SListElem *next;
};

class SList
{
public:
    SList(): h(0) {} // 0 denotes empty list
    ~SList() { Release(); } // destruction
    void Prepend(char c);
    void Del();
    SListElem *First() const { return h; }
    void Print() const;
    void Release();

private:
    SListElem *h; // head of SList
};
```