The this Pointer

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

**Example:**

```cpp
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;
};
```

The Inheritance Chain

When a method in a derived class uses `this`, it refers to the instance of the derived class.

```cpp
class Derived : public Base
{
public:
    void print() const { std::cout << this << '
'; }

private:
    Derived() : Base(0) { }
};
```

```
Base* base_ptr = new Derived();
Derived derived = base_ptr;  // Reference to derived object
Base* derived_ptr = dynamic_cast<Base*>(base_ptr);  // Downcast to base object
```

**static and const Members**

- The const and static member function implementation can be understood in 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.

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**

- 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.

**Destructors**

- cannot take arguments,
- cannot be overloaded.

A destructor is invoked implicitly whenever an object goes out of scope.

Constructors and destructors do not have return types and cannot use `return expression` statements.

Classes with Constructors

**Example:** A data type ModInt for storing numbers that are computed with a modulus.

```cpp
class ModInt
{
public:
    ModInt(int i);  // constructor declaration
    void assign(int i) { v = i % ModInt::modulus; }
    void print() const { cout << v << " \n"; }
    const static int modulus;

private:
    int v;
};
```

```cpp
const int ModInt::modulus = 60;
```

```cpp
ModInt ModInt(int i) { v = i % ModInt::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

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];
};
```

```cpp
const char* PrintChar::rep[128] = {"nul", "soh", "stx", ..., "", "", "del"};
```

**Example:**

```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.

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

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

Member access is correct in all three cases.
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.

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(); }
  void Prepend(char c);
  void Del();
  SListElem *First() const { return h; }
  void Print() const;
  void Release();
private:
  SListElem *h; // head of SList
};

void SList::Prepend(char c)
{
  SListElem *temp = new SListElem; // create new element
  assert( temp != 0);
  temp->next = h; // link to SList
  temp->data = c;
  h = temp; // update head of SList
}

void SList::Del()
{
  SListElem *temp = h;
  if (temp != 0)
  {
    h = h->next; // let head point to 2nd element
    delete temp; // delete 1st element
  }
}

void SList::Print() const
{
  SListElem *temp = h;
  while (temp != 0) // detect end of Slist
  {
    cout << temp->data << " -> ";
    temp = temp->next; // temp proceeds to next element
  }
  cout << "NULL\n";
}
```
A Singly Linked List

void SList::Release()
{
    while (h != 0) // as long as there are elements...
        Del();  // ... delete them.
}

A Singly Linked List

int main()
{
    SList MyList;
    MyList.Insert('S');
    MyList.Insert('T');
    MyList.Insert('A');
    MyList.Insert('R');
    MyList.Insert('T');
    MyList.Delete();
    MyList.Print();
}

Output: R -> A -> T -> S -> NULL

A Two-Dimensional Array

- We are now going to implement a dynamic and safe two-dimensional array.
- We will use a pointer-to-pointer-to-base type structure for this implementation.

A Two-Dimensional Array

class Matrix
{
    public:
        Matrix(int sizeX, int sizeY);
        ~Matrix();
        int GetSizeX() const { return dx; }
        int GetSizeY() const { return dy; }
        long &Element(int x, int y); // return reference to an element
        void Print() const;
    private:
        long **p; // pointer to a pointer to a long integer
        int dx, dy;
};

Matrix::Matrix(int sizeX, int sizeY)
{
    assert(sizeX > 0 && sizeY > 0);
    p = new long*[dx]; // create array of pointers to long integers
    for (int i = 0; i < dx; i++)
        p[i] = new long[dy]; // for each pointer, create array of long integers
    assert(p[i] != 0);
    for (int j = 0; j < dy; j++)
        p[i][j] = 0;
}

Matrix::~Matrix()
{
    for (int i = 0; i < dx; i++)
        delete [] p[i]; // delete arrays of long integers
    delete [] p; // delete array of pointers to long integers
}

long &Matrix::Element(int x, int y)
{
    assert(x >= 0 && x < dx && y >= 0 && y < dy);
    return p[x][y];
}
A Two-Dimensional Array

void Matrix::Print() const
{
    cout << endl;
    for (int y = 0; y < dy; y++)
    {
        for (int x = 0; x < dx; x++)
            cout << p[x][y] << "\t";
        cout << endl;
    }
}

A Two-Dimensional Array

int main()
{
    Matrix MyMatrix(3, 5);
    MyMatrix.Element(1, 1) = 88;
    MyMatrix.Element(2, 2) = 11;
    MyMatrix.Element(2, 4) = MyMatrix.Element(1, 1) + MyMatrix.Element(2, 2);
    cout << MyMatrix.Element(2, 4) << endl;
    MyMatrix.Print();
}

Output:

99
0 0 0
0 88 0
0 0 11
0 0 0
0 0 99