

# Case Study

## CS 201

*This slide set covers pointers and arrays in C++. You should read Chapters 9 and 10 from your Deitel & Deitel book.*

# IntArray class

- Let's implement an `IntArray` class to represent an array of integers
- This class will have the following features:
  - The array knows its size
  - Array items do not have garbage values
  - It always handles allocation and deallocation (such that as a user, you do not need to worry about explicitly using the `new` and `delete` operators)
  - It always deep copies the array items when an array's copy is needed or the assignment operator is applied on the array
  - The subscript operator has array bound checking for accessing the array items

# IntArray definition

```
class IntArray {
public:
    // If a class has dynamically allocated data members, it is highly recommended to
    // re-implement the destructor and copy constructor, and overload the assignment
    // operator (instead of using the ones provided by the compiler)

    IntArray( const int = 0 );           // constructor with a default argument
    IntArray( const IntArray& );        // copy constructor
    ~IntArray();                        // destructor
    IntArray& operator=( const IntArray& ); // overloaded assignment operator
    int& operator[]( const int );      // overloaded subscript operator

private:
    int size;                          // number of array items
    int* data;                         // dynamically allocated array
};
```

# IntArray implementation (*constructor*)

- Constructors are called when an object is constructed (either by declaration or using the `new` operator)
  - The storage class of an object determines when it is constructed
- It is possible to implement multiple constructors as long as their signatures are different
  - The compiler selects which one to use based on its argument(s)
  - If an object is initialized with another object of the same type at its construction, the compiler calls the copy constructor

```
// Constructor with a single integer
// parameter (this parameter has a
// default value set in the class
// definition such that it also serves
// as a default constructor)
IntArray::IntArray( const int aSize ) {
    if ( aSize <= 0 ) {
        size = 0;
        data = nullptr;
    }
    else {
        size = aSize;
        data = new int[ size ];
        for ( int i = 0; i < size; i++ )
            data[ i ] = 0;
    }
}
```

# IntArray implementation (*copy constructor*)

- Copy constructor is called when
  - An object is initialized with another of the same type at its declaration (construction)  
`IntArray prev;`  
`IntArray current( prev );`  
`IntArray next = current;`
  - An object is passed by value as an argument to a function (pass-by-value)
  - An object is returned by value from a function (however, C++ Standard allows compilers to optimize this)
- If not provided explicitly, the compiler provides a default copy constructor performing memberwise shallow copy
  - It does not deep copy the data members

```
// This copy constructor deep copies the
// array items as opposed to default copy
// constructor provided by the compiler
IntArray::IntArray( const IntArray& arr )
    : size( arr.size ) {
    if ( size > 0 ) {
        data = new int[ size ];
        for ( int i = 0; i < size; i++ )
            data[ i ] = arr.data[ i ];
    }
    else
        data = nullptr;
}
```

*It must receive its argument by reference (not by value). Otherwise, it results in infinite recursion.*

*Its argument should also be const to allow a constant object to be copied and to be used only as an rvalue inside the function.*

# IntArray implementation (*destructor*)

- Destructor is a special member function that is called implicitly when an object is destructed (either when its lifetime ends or when the `delete` operator is used)
  - Destructor calls are usually made in the reverse order of their corresponding constructor calls
  - However, the storage class of objects may alter this order
- Each class should have only one destructor (no overloading is allowed)
- If not provided explicitly, the compiler creates an “empty” destructor

```
IntArray::~~IntArray() {  
    if ( data )  
        delete[] data;  
}
```

```
// Constructor call for a single  
// dynamically created object  
IntArray* a1 = new IntArray( 400 );  
  
// Default constructor call for every  
// object in the array  
IntArray* a2 = new IntArray[ 5 ];  
  
// Destructor call for the single object  
delete a1; //  
  
// Every object in the array receives a  
// destructor call. If "delete a2;" is  
// used, only the first object receives  
// a destructor call  
delete[] a2;
```

# Operator overloading

- For every class, the following operators are provided by the compiler
  - Assignment operator (=) → performs memberwise assignment between two objects
  - Address operator (&) → returns the address of an object
- Although they are provided by the compiler, these operators can also be overloaded by the programmer
- Other operators can also be overloaded except `.` `::` `?:` `sizeof`
- Operator overloading should be done for a class individually
  - By defining a member function (in that class) for this operator
  - Where the name of this function should be **`operator <operator-to-be-overloaded>`**

# IntArray implementation (*assignment operator*)

- It is called when the left operand is an object
- If not provided explicitly, the compiler provides a default assignment operator that assigns each data member of the right object to the same data member of the left object
- However, this default assignment operator performs shallow copy for the memberwise assignments

One can also define additional assignment operators where the right operand is of another data type

```
IntArray& IntArray::operator=( const IntArray& right ) {  
    if ( &right != this ) { // to avoid self-assignment  
        if ( size != right.size ) {  
            if ( size > 0 )  
                delete[] data;  
            size = right.size;  
            if ( size > 0 )  
                data = new int[ size ];  
            else  
                data = nullptr;  
        }  
        for ( int i = 0; i < size; i++ )  
            data[ i ] = right.data[ i ];  
    }  
    return *this; // to allow cascading  
}
```



# this pointer

- Every object has access to its own address through a pointer called this
  - The `this` pointer is not a part of the object
  - The compiler passes it as an implicit argument to a `non-static` member function call of this object
- An object uses its `this` pointer
  - Implicitly when accessing its members directly
  - Explicitly when using the `this` keyword
- The type of the `this` pointer depends on the object's type and whether the executing member function is declared as `const`

## static member functions

- A member function can be declared as `static` if it does not access any `non-static` data member or call any `non-static` member function of its class
- `static` data members of a class exist in memory and its `static` member functions can be called even when there exist no object of this class in memory
- A `static` member function does not have the `this` pointer

# IntArray implementation (*subscript operator*)

- Other operators can also be overloaded
- This subscript operator facilitates array bound checking for accessing the array items

The return type of this function should be of a reference type since its returned value can be used both as **an lvalue** and as an rvalue

```
IntArray arr( 5 );  
cout << arr[ 3 ]; // used as an rvalue  
arr[ 3 ] = 10; // used as an lvalue
```

```
int& IntArray::operator[]( const int ind ){  
    if ( ind < 0 || ind >= size )  
        throw out_of_range("Invalid index");  
    else  
        return data[ ind ];  
}
```

```
// You can throw and catch exceptions also  
// in C++, as in the following example  
#include <exception>  
int main(){  
    IntArray arr(100);  
    try {  
        arr[130] = 20;  
    }  
    catch ( const exception& e ){  
        cout << e.what() << endl;  
    }  
    return 0;  
}
```

**Example:** Given the following `Test` class, what are the outputs of the following programs?

These examples are for you to better understand when the constructor, copy constructor, destructor, and assignment operator are called.

```
class Test {
public:
    Test( int i = 0 ){
        id = i;
        cout << "Constructor " << id << endl;
    }
    ~Test(){
        cout << "Destructor " << id << endl;
    }
    Test( const Test& o ){
        id = o.id;
        cout << "Copy const " << id << endl;
    }
    Test& operator=( const Test& right ){
        id = right.id;
        cout << "Assignment " << id << endl;
        return *this;
    }
    int id;
};
```

**Example:** Given the `Test` class above, what are the outputs of this program?

Do not forget that the constructor, copy constructor, destructor, and assignment operator are called only **for an object**. For example, they are not called for an object pointer or a class data member (unless this data member is an object of another class).

```
Test t1( 10 );
Test t2( 20 );

void foo( bool flag ){
    Test t3( 30 );
    static Test t4( 40 );

    if ( flag ){
        Test t5( 50 );
        Test t6( 60 );
    }
    Test t7( 70 );
}

int main() {

    cout << "checkpoint 1---" << endl;
    Test t8( 80 );

    cout << "checkpoint 2---" << endl;
    foo( false );

    cout << "checkpoint 3---" << endl;
    foo( true );

    cout << "checkpoint 4---" << endl;
    return 0;
}
```

**Example:** Given the `Test` class above, what are the outputs of this program?

Do not forget that the constructor, copy constructor, destructor, and assignment operator are called only **for an object**. For example, they are not called for an object pointer or a class data member (unless this data member is an object of another class).

```
int main() {  
  
    cout << "checkpoint 1---" << endl;  
    Test *b1;  
  
    cout << "checkpoint 2---" << endl;  
    b1 = new Test ( 100 );  
    delete b1;  
  
    cout << "checkpoint 3---" << endl;  
    b1 = new Test [2];  
    b1[0].id = 200;  
    b1[1].id = 300;  
    delete[] b1;  
  
    cout << "checkpoint 4---" << endl;  
    b1 = new Test [2];  
    b1[0].id = 400;  
    b1[1].id = 500;  
    delete b1;  
  
    cout << "checkpoint 5---" << endl;  
    return 0;  
}
```

**Example:** Given the `Test` class above, what are the outputs of this program?

Do not forget that the constructor, copy constructor, destructor, and assignment operator are called only **for an object**. For example, they are not called for an object pointer or a class data member (unless this data member is an object of another class).

```
void bar ( Test a, Test* b, Test& c ) {  
    // ...  
}  
  
int main() {  
  
    cout << "checkpoint 1---" << endl;  
    Test t1(11);  
    Test& t2 = t1;  
    Test t3 = t1;  
    t3.id = 33;  
  
    cout << "checkpoint 2---" << endl;  
    bar( t1, &t2, t3 );  
  
    cout << "checkpoint 3---" << endl;  
    Test* t4;  
    Test* t5;  
    t4 = &t1;  
    t5 = t4;  
    *t4 = t1;  
  
    cout << "checkpoint 4---" << endl;  
    bar( *t5, &t1, *t4 );  
  
    cout << "checkpoint 5---" << endl;  
    return 0;  
}
```