Arrays, Pointers and References
Arrays of objects of class can be declared just like other variables.

- class A{ ... };
- A ob[4];
- ob[0].f1();  // let f1 is public in A
- ob[3].x = 3; // let x is public in A

In this example, all the objects of the array are initialized using the default constructor of A.

If A does not have a default constructor, then the above array declaration statement will produce compiler error.
Arrays of Objects

- If a class type includes a constructor, an array of objects can be initialized.
- Initializing array elements with the constructor taking an integer argument

```cpp
class A { public: int a; A(int n) { a = n; } }
```

- `A ob[2] = { A(-1), A(-2) };`
- `A ob2[2][2] = { A(-1), A(-2), A(-3), A(-4) };`

- In this case, the following shorthand form can also be used

```cpp
A ob[2] = { -1, -2 };`
If a constructor takes two or more arguments, then only the longer form can be used.

```cpp
class A { public: int a, b; A(int n, int m) { a = n; b = m; } };
```

- `A ob[2] = { A(1, 2), A(3, 4) };`
- `A ob2[2][2] = { A(1, 1), A(2, 2), A(3, 3), A(4, 4) };`
ARRAYS OF OBJECTS

- We can also mix no argument, one argument and multi-argument constructor calls in a single array declaration.

```cpp
class A
{
 public:
  A() { ... } // must be present for this example to be compiled
  A(int n) { ... }
  A(int n, int m) { ... }
};

A ob[3] = { A(), A(1), A(2, 3) };
```
**Using Pointers to Objects**

- We can take the address of objects using the address operator (`&`) and store it in object pointers.
  - `A ob; A *p = &ob;`
- We have to use the arrow (`->`) operator instead of the dot (`.`) operator while accessing a member through an object pointer.
  - `p->f1(); // let f1 is public in A`
- Pointer arithmetic using an object pointer is the same as it is for any other data type.
  - When incremented, it points to the next object.
  - When decremented, it points to the previous object.
**THIS POINTER**

- A special pointer in C++ that points to the object that generates the call to the method.

- Let,
  - `class A{ public: void f1() { ... } };`
  - `A ob; ob.f1();`

- The compiler automatically adds a parameter whose type is “pointer to an object of the class” in every non-static member function of the class.

- It also automatically calls the member function with the address of the object through which the function is invoked.

- So the above example works as follows –
  - `class A{ public: void f1( A *this ) { ... } };`
  - `A ob; ob.f1( &ob );`
**THIS POINTER**

- It is through this pointer that every non-static member function knows which object’s members should be used.

```cpp
class A {
    int x;
public:
    void f1() {
        x = 0; // this->x = 0;
    }
};
```
**This Pointer**

- This pointer is generally used to access member variables that have been hidden by local variables having the same name inside a member function.

```cpp
class A{
    int x;
public:
    A(int x) {
        x = x;  // only copies local ‘x’ to itself; the member ‘x’ remains uninitialized
        this->x = x;  // now its ok
    }
    void f1() {
        int x = 0;
        cout << x;  // prints value of local ‘x’
        cout << this->x;  // prints value of member ‘x’
    }
};
```
Using new and delete

C++ introduces two operators for dynamically allocating and deallocating memory:

- \( p\_var = \textit{new type} \)
  - new returns a pointer to dynamically allocated memory that is sufficient to hold a data object of type \textit{type}
- \( \textit{delete} \ p\_var \)
  - releases the memory previously allocated by new

- Memory allocated by new must be released using delete
- The lifetime of an object is directly under our control and is unrelated to the block structure of the program
**Using new and delete**

- In case of insufficient memory, `new` can report failure in two ways
  - By returning a null pointer
  - By generating an exception
- The reaction of `new` in this case varies from compiler to compiler
**Using new and delete**

- Advantages
  - No need to use `sizeof` operator while using `new`.
  - New automatically returns a pointer of the specified type.
  - In case of objects, new calls dynamically allocates the object and call its constructor.
  - In case of objects, delete calls the destructor of the object being released.
**Using new and delete**

- Dynamically allocated objects can be given initial values.

  - `int *p = new int;`
    - Dynamically allocates memory to store an integer value which contains garbage value.
  
  - `int *p = new int(10);`
    - Dynamically allocates memory to store an integer value and initializes that memory to 10.
    - *Note the use of parenthesis () while supplying initial values.*
**Using new and delete**

- `class A{ int x; public: A(int n) { x = n; } };`
  - `A *p = new A(10);`
    - Dynamically allocates memory to store a A object and calls the constructor A(int n) for this object which initializes x to 10.
  - `A *p = new A;`
    - It will produce **compiler error** because in this example class A does not have a default constructor.
**Using new and delete**

- We can also create dynamically allocated arrays using `new`.
- But deleting a dynamically allocated array needs a slight change in the use of `delete`.
- *It is not possible to initialize an array that is dynamically allocated.*
  - `int *a= new int[10];`
    - Creates an array of 10 integers
    - All integers contain garbage values
    - *Note the use of square brackets []*
  - `delete [] a;`
    - Delete the entire array pointed by `a`
    - *Note the use of square brackets []*
**USING NEW AND DELETE**

- It is not possible to initialize an array that is dynamically allocated, in order to create an array of objects of a class, the class must have a default constructor.

```cpp
class A {
    int x;
public:
    A(int n) { x = n; }
};

A *array = new A[10]; // compiler error
```

```cpp
class A {
    int x;
public:
    A() { x = 0; }
    A(int n) { x = n; }
};

A* array = new A[10]; // no error
// use array
delete [ ] array;
```
USING NEW AND DELETE

- A *array = new A[10];
  - The default constructor is called for all the objects.
- delete [] array;
  - Destructor is called for all the objects present in the array.
REFERENCES

- A reference is an implicit pointer
- Acts like another name for a variable
- Can be used in three ways
  - A reference can be passed to a function
  - A reference can be returned by a function
  - An independent reference can be created
- Reference variables are declared using the & symbol
  - void f(int &n);
- Unlike pointers, once a reference becomes associated with a variable, it cannot refer to other variables
REFERENCES

- Using pointer -
  ```
  void f(int *n) {
    *n = 100;
  }
  void main() {
    int i = 0;
    f(&i);
    cout << i; // 100
  }
  ```

- Using reference -
  ```
  void f(int &n) {
    n = 100;
  }
  void main() {
    int i = 0;
    f(i);
    cout << i; // 100
  }
  ```
A reference parameter fully automates the call-by-reference parameter passing mechanism

- No need to use the address operator (&) while calling a function taking reference parameter

- Inside a function that takes a reference parameter, the passed variable can be accessed without using the indirection operator (*)
REFERENCES

- Advantages
  - The address is automatically passed
  - Reduces use of ‘&’ and ‘*’
  - When objects are passed to functions using references, no copy is made
    - Hence destructors are not called when the functions ends
    - Eliminates the troubles associated with multiple destructor calls for the same object
**PASSING REFERENCES TO OBJECTS**

- We can pass objects to functions using references

- No copy is made, destructor is not called when the function ends

- As reference is not a pointer, we use the dot operator (.) to access members through an object reference
**Passing References to Objects**

class myclass {
    int x;
public:
    myclass() {
        x = 0;
        cout << "Constructing\n";
    }
    ~myclass() {
        cout << "Destructing\n";
    }
    void setx(int n) { x = n; }
    int getx() { return x; }
};

void main() {
    myclass obj;
    cout << obj.getx() << endl;
    f(obj);
    cout << obj.getx() << endl;
}

Output:
    Constructing
    0
    500
    Destructing
RETURNING REFERENCES

- A function can return a reference
- Allows a function to be used on the left side of an assignment statement
- But, the object or variable whose reference is returned must not go out of scope
- So, we should not return the reference of a local variable
  - For the same reason, it is not a good practice to return the pointer (address) of a local variable from a function
int x; // global variable
int &f() {
    return x;
}
void main() {
    x = 1;
    cout << x << endl;
    f() = 100;
    cout << x << endl;
    x = 2;
    cout << f() << endl;
}

Output:
1
100
2
So, here f() can be used to both set the value of x and read the value of x

Example: From Book(151 – 153)
INDEPENDENT REFERENCES

- Simply another name for another variable
- Must be initialized when it is declared
  - `int &ref; // compiler error`
  - `int x = 5; int &ref = x; // ok`
  - `ref = 100;`
  - `cout << x; // prints “100”`
- An independent reference can refer to a constant
  - `int &ref=10; // compile error`
  - `const int &ref = 10;`
RESTRICTIONS

- We cannot reference another reference
  - Doing so just becomes a reference of the original variable
- We cannot obtain the address of a reference
  - Doing so returns the address of the original variable
  - Memory allocated for references are hidden from the programmer by the compiler
- We cannot create arrays of references
- We cannot reference a bit-field
- References must be initialized unless they are members of a class, are return values, or are function parameters
Lecture Contents

- Teach Yourself C++
  - Chapter 4 (See All Exercise)