C++ overloaded operators

- any inbuilt operator can be overloaded to add a new, additional meaning

- 2 ways to do this

1. add a method to a class

e.g.
```
class C {
public:
    C operator+ (C& obj) {
        C* temp = new C();
          :
          :
        return *temp;  // returns a copy
    }
    . . .
};
```

```
C c1, c2, c3;
c2.init(...);
c3.init(...);
```

$$c1 = \underbrace{c2 + c3};$$

translated into `c2.operator+(c3);`

2. use a global function and a friend declaration

```
class C {
    ...
    friend C operator+(C& obj1, C& obj2) ;
};
```

global function

```
C operator+(C& obj1, C& obj2) {
    C* temp = new C();
    :
    return *temp;
}
```

same example applies :

$$c1 = \underbrace{c2 + c3}_{\text{operator+(c2, c3);}};$$

This has an advantage, because we can overload using inbuilt types as well as class types

e.g.
```
C operator+(int & x, C & obj1) {
    ...
}
```

A complete example : dynamic array

- an array which can grow in size
- we will encapsulate a pointer to the first element of the array
- accesses to array elements will test for index value and adjust the array size if necessary
- we will have to choose the element type (Later we will turn it into a template)
- it will use references, destructor function, overloaded operators

Sketch :

```
class DynArray {
private:
    int *arr;
    int len;
public:
    DynArray (int len) {
        arr = new int [len];
        this -> len = len;
        memset (arr, 0, len * sizeof (int));
    }
    ...
};
```

Use :

```
DynArray da1 (10);
```

To avoid memory leakage, add a destructor

```
public:
    ~DynArray() {                  | destructor
        delete [] arr;             |
    }                              |
    int length() {                 | accessor for len
        return len;                |
    }                              |
```

Now the overloaded operator

e.g. `da1[1] = 3;`

sketch:

```
int& operator[](int ind) {
    return arr[ind];
}
```

Returning a reference to an integer ensures that the operator can be used on LHS of an assignment.

It will also work for RHS

```
DynArray da1(10);
da1[0] = 3;
   ↑
returns a ref. to first element
int x;
x = da1[0];
```

We need to test for index values out of range and adjust the size of array accordingly.

```
int & operator[] (int ind) {
   if (ind >= len) {
      int* newarr = new int [ind+1];
      memcpy (newarr, arr, len * sizeof(int));
      memset (newarr + len, 0, (ind - len +) *
                                   sizeof(int));
      delete [] arr;
      arr = newarr;                    return arr[ind];
      len = ind +1;                 ↙
```

```
DynArray da1(10);
da1[0]= 3;
da1[4] = 32;
da1[20] = 10;
da1[100000] =20;
```

DynArray has one problem: -ve index values

```
DynArray da1(10);
da1[-1] = 3; ???
```

To handle this properly we will use exceptions (could test for -ve values a do nothing)

Exceptions are handled with try/catch/throw mechanism.

```
try {
    ....
}
catch (...) {
    ...
}
```

try block does something that cause an exception to be thrown

type spec. for type of thrown value

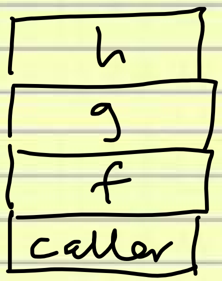
e.g.

```
try {
    f();
}
catch (int e) {
    cout << e << endl;
}
```

```
void f() {
    ...
    if (x == 0) {
        throw 1001;
    }
    ...
}
```

```
try {
    f() → g() → h() → throws int
}
catch (int e) {
     ...
 }
```

Look at stack :



When h throws a value it immediately returns, popping the stack. The g returns, then f returns and we back with the caller on top-of-stack.

When a function returns, its locals are destroyed in normal fashion.

Also we can have multiple catchers

```
try {
  ...
}
catch (int e) {
  ..
}
catch (Exception1 e) {
  ...
}
catch (Exception2 e) {
  ...
}
```

```
catch (...) {
  ...
}
```

will catch anything

In DynArray, we add a test for -ve index values

```
int & operator[](int ind) {
    if (ind < 0) {
        throw 32;
    }
    .... as before
}
```

Use:

```
try {
    DynArray d1(10);
    d1[-2] = 2;
}
catch (int e) {
    if (e == 32) {
        cout << "-ve index" << endl;
    }
}
```

Templates

- a template is a form (class or a global function) parameterized with a type

```
template <class T>
class DynArray {
private:
 T ~~int~~* arr;
   int len;
public:
    DynArray(int len) {
        ...
    }
   T ~~int~~& operator(int ind) {
        ...
    }
    ~DynArray() { ... }
};
```

Use :

```
DynArray<int> d1(10);
DynArray<float> d2(10);
DynArray<C> d3(10);
            ↑
       class type
```