Vectors and Strings

Vectors

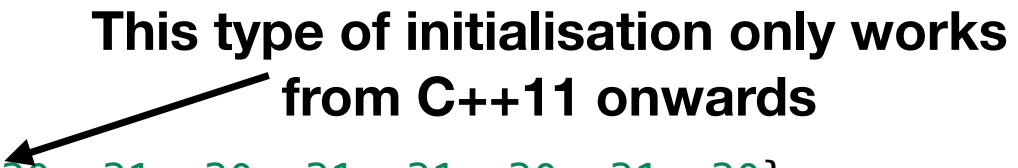
- Built-in C++ dynamic arrays
  - Size can be dynamically increased
- Contiguous storage
- Storage is handled internally, not visible to the user.
  - Example of Interface VS Implementation

Vectors: Example

```cpp
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> daysInMonth = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30};
    for (auto i : daysInMonth)
    {
        cout << i << endl;
    }
    daysInMonth.push_back(31);
    cout << "Using iterator\n";
    vector<int> :: iterator i;
    for (i = daysInMonth.begin(); i != daysInMonth.end(); i++)
    {
        cout << *i << endl;
    }
}
```

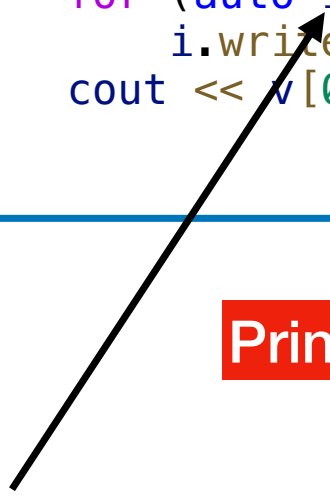


Vector of objects

```
class IntCell
{
    public:
        explicit IntCell(int initialValue=0)
            : storedValue(initialValue) {}
        int read() const {return storedValue;}
        void write(int x) {storedValue = x;}
    private:
        int storedValue;
};
```

```
int main()
{
    vector<IntCell> v(10);
    for (auto i : v)
        i.write(100);
    cout << v[0].read() << endl;
}
```

Prints 0

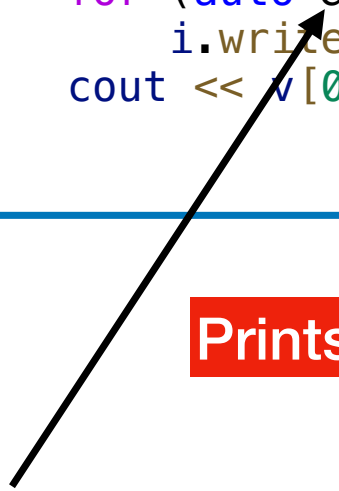


i is a copy of vector element

Vector of objects

```
class IntCell
{
    public:
        explicit IntCell(int initialValue=0)
            : storedValue(initialValue) {}
        int read() const {return storedValue;}
        void write(int x) {storedValue = x;}
    private:
        int storedValue;
};
```

```
int main()
{
    vector<IntCell> v(10);
    for (auto & i : v)
        i.write(100);
    cout << v[0].read() << endl;
}
```



Prints 100

i is a reference to vector element

Strings

- Built-in C++ character arrays.
  - Dynamically sized.
- Provides support for various operators: concatenation (+), string comparison (==,<), etc.
- Provides a number of useful helper functions: size, insert, find, remove, etc.
- Both string and vector classes have pre-defined copy/move constructors and assignment operators.

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string str = "Welcome to OOAIA Lab";
    string t = str + "!!!";
    cout << t << endl;
    for (auto c : str)
        cout << c << endl;
}
```

Overloading

Introduction

- Oftentimes, we have multiple functions conceptually performing the same task.
  - ++ is used for incrementing integers, floats, doubles, iterators.
  - Indexing operator ([ ]) used for arrays, vectors, strings.
  - + is used for adding integers, floats, doubles, concatenating strings.
- Overloading allows us to use the same function name/operator for conceptually similar tasks.
  - Can tremendously improve code readability.
  - Also called Polymorphism.

Function Overloading: Example

```
class IntCell
{
    public:
        explicit IntCell(int initialValue=0)
            : storedValue(initialValue) {}
        int read() const {return storedValue;}
        void write(int x) {storedValue = x;}
        void add(int x) {storedValue += x;}
        void add(int x, int y) {storedValue += x + y;}
    private:
        int storedValue;
};
```

```
int main()
{
    IntCell i(10);
    i.add(10);
    i.add(10,10);
    cout << i.read() << endl;
}
```

Prints 40

Overloading a unary operator

```
class IntCell
{
public:
    explicit IntCell(int initialValue=0)
        : storedValue(initialValue) {}
    int read() const {return storedValue;}
    void write(int x) {storedValue = x;}
    void operator ++ () {storedValue++;}
private:
    int storedValue;
};
```

```
int main()
{
    IntCell i(10);
    ++i;
    cout << i.read() << endl;
}
```

Prints 11

Prefix and postfix increment

```
class IntCell
{
public:
    explicit IntCell(int initialValue=0)
        : storedValue(initialValue) {}
    int read() const {return storedValue;}
    void write(int x) {storedValue = x;}
    IntCell operator ++ ()
      {return IntCell(++storedValue);}
    IntCell operator ++ (int)
      {return IntCell(storedValue++);};
private:
    int storedValue;
};
```

```
int main()
{
    IntCell i(10);
    IntCell j = ++i;
    IntCell k = i++;
    cout << i.read() << " " <<
    j.read() << " " << k.read() << endl;
}
```

Prints 12 11 11

Overloading a binary operator

```
class IntCell
{
public:
    explicit IntCell(int initialValue=0)
        : storedValue(initialValue) {}
    int read() const {return storedValue;}
    void write(int x) {storedValue = x;}
    IntCell operator ++ ()
       {return IntCell(++storedValue);}
    IntCell operator ++ (int)
       {return IntCell(storedValue++);};
IntCell operator + (IntCell x)
      {return IntCell(storedValue + x.read());}
private:
    int storedValue;
};
```

```
int main()
{
    IntCell i(10), j(20);
    IntCell k = i + j;
    cout << k.read() << endl;
}
```

Prints 30

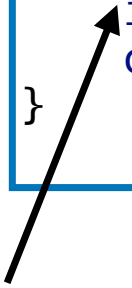
Translates to i.operator+(j)

The object to the left of + (i.e. i) becomes the receiver object, while the object to the right becomes the argument.

Overloading a binary operator

```
class IntCell
{
public:
    explicit IntCell(int initialValue=0)
        : storedValue(initialValue) {}
    int read() const {return storedValue;}
    void write(int x) {storedValue = x;}
    IntCell operator ++ ()
       {return IntCell(++storedValue);}
    IntCell operator ++ (int)
       {return IntCell(storedValue++);};
    void operator + (IntCell x)
      {return storedValue += x.read());}
private:
    int storedValue;
};
```

```
int main()
{
    IntCell i(10), j(20);
    i + j;
    cout << i.read() << endl;
}
```



Translates to i.operator+(j)

Prints 30

Overloading <<

- Suppose we want to overload << for `IntCell` such that `cout << c` (for `IntCell` variable c) nicely prints c's contents.
  - Relevant info: `cout` is an object of type `std::ostream`.
- We would have to overload << inside the `ostream` class to handle objects of type `IntCell`.
  - We cannot do this for every user-defined type.

Overloading <<

```
class IntCell
{
    ...
};
ostream & operator << (ostream & out, IntCell & c)
{
    return out << "(IntCell " << c.read() << ")";
}
int main()
{
    IntCell i(10), j(20);
    IntCell k = i + j;
    cout << k << endl;
}
```

Overloading outside class

Prints (IntCell 30)

Overloading >>

```
class IntCell
{
    ...
};

istream & operator >> (istream & in, IntCell & c){
    int v;
    in >> v;
    c.write(v);
    return in;
}
ostream & operator << (ostream & out, IntCell & c)
{
    return out << "(IntCell " << c.read() << ")";
}
int main()
{
    IntCell c;
    cin >> c;
    cout << c << endl;
}
```

Using `friend`

```
class IntCell
{
    ...
    friend istream & operator >> (istream & in, IntCell & c);
};

istream & operator >> (istream & in, IntCell & c){
     return in >> c.storedValue;
}
ostream & operator << (ostream & out, IntCell & c)
{
    return out << "(IntCell " << c.read() << ")";
}
int main()
{
    IntCell c;
    cin >> c;
    cout << c << endl;
}
```

Can access `private` members in `friends`

Overloading [ ]

```
class safeArray
{
    private:
        int * arr;
        int capacity;
    public:
        safeArray(int inCapacity) : capacity(inCapacity){
            arr = new int[capacity];
        }
        int & operator [] (int index){
            if (index >= 0 && index < capacity)
                return arr[index];
            else
                {cout << "\n Index out of bounds \n"; exit(1);}
        }
};
```

```
int main()
{
    safeArray arr(50);
    arr[4] = 10;
    cout << arr[4] << endl;
    cout << arr[50] << endl;
}
```

Prints

10

Index Out of bounds

Type Conversion

```
class IntCell
{
public:
    explicit IntCell(int initialValue=0)
        : storedValue(initialValue) {}
    int read() const {return storedValue;}
    void write(int x) {storedValue = x;}
    operator int () const { return storedValue;}
private:
    int storedValue;
};
```

```
int main()
{
    IntCell c(50);
    int i = c;
    cout << i << endl;
}
```

Type Conversion Operator, implicitly called here