UNSW SYDNEY

MTRN2500

Tutorial 4 – Classes II & STL

Classes II

UNSW
SYDNEY

Specialty Constructors

- We introduced some basic constructors last week
- Some additional specialty constructors exist for different interactions with classes you write
  - Copy constructor – Constructs an object that is a copy of an existing one
  - Move constructor – Constructs an object that has the state of an existing one which is ‘deactivated’

Copy Constructor

```
1. #include <iostream>
2. #include <string>

3. class StudentInfo {
4. public:
5.     StudentInfo(int ID, const std::string &name)
6.         : mID {ID}, mName {name} {
7.     }

8.     StudentInfo(const StudentInfo &other)
9.         : mID {other.mID}, mName {other.mName} {
10.        std::cout << "Copy constructor is called.\n";
11.    }

12.    void print() {
13.        std::cout << "mID: " << mID << ", mName: " << mName << '\n';
14.    }
15.private:
16.    int mID {};
17.    std::string mName {};
18.};
```

```
1. int main() {

2.     StudentInfo s1 {101, "Emma"};
3.     StudentInfo s2 {s1};
4.     s1.print();
5.     s2.print();
6.
       return 0;
7. }
```

Move Constructor

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

class StudentInfo {
public:
    StudentInfo(int ID, const std::string &name)
        : mID {ID}, mName {name} {
    }

    StudentInfo(StudentInfo &&other) noexcept
        : mID{std::exchange(other.mID, 0)}
        , mName{std::exchange(other.mName, {})} {
        std::cout << "Move constructor is called.\n";
    }

    void print() {
        std::cout << "mID: " << mID << ", mName: " << mName << '\n';
    }
private:
    int mID {};
    std::string mName {};
};
```

```cpp
int main() {

    StudentInfo s1 {101, "Emma"};
    StudentInfo s2 std::move{s1};
    s2.print();

    return 0;
}
```

Copy and Move Assignment

- As well as copy and move constructors, assignment operators can be implemented for these actions
- These are used to assign value to already existing objects
- Review lectures for their implementation

Static Class Members

- Static members are belonging to an overall class rather than a particular instance (object)
- There is always one copy of a static member per-class no matter how many instances
- Can be accessed outside of an object or within
- Can have either static data members of methods

Static Class Members – Continued

```
#include <iostream>

class Cube {
    public:
        // Constructor definition
        Cube(double sideLength = 2.0)
            : mSideLength {sideLength} {
                std::cout <<"Constructor called.\n";
                // Increase every time object is created
                objectCount++;
        }
        // creating a static function that
            returns static data member
        static int getCount() {
            return objectCount;
        }
    private:
        static int objectCount;
        double mSideLength {};     // Side of a cube
};
```

```
// Initialize static member of class Cube
int Cube::objectCount {0};

int main() {
    Cube cube1(5.0);    // Declare cube1
    Cube cube2(8.0);    // Declare cube2
    Cube cube3(10.0);   // Declare cube3

    // Print total number of objects.
    std::cout << "Total objects: " <<
  Cube::getCount();

    return 0;
}
```

UNSW SYDNEY

Operator Overloading

- Operators (+, -, <<, >>, +=, etc.) are widely used when writing software
- We can define custom functionality for these for user-defined objects
- We overload operators for this functionality which act as a function call

Common Operators

Common operators					
assignment	increment decrement	arithmetic	logical	comparison	member access
a = b a += b a -= b a *= b a /= b a %= b a &= b a |= b a ^= b a <<= b a >>= b	++a --a a++ a--	+a -a a + b a - b a * b a / b a % b ~a a & b a | b a ^ b a << b a >> b	!a a && b a || b	a == b a != b a < b a > b a <= b a >= b a <=> b	a[...] *a &a a->b a.b a->*b a.*b

Operator Overload Example

```
1. #include <iostream>

2. class NewInt {
3. public:
4.     NewInt(int num) : mInt{num} {}
5.     int getInt() const {return mInt;}
6.
7.     NewInt& operator+=(const NewInt& rhs) {
  //binary operator overloading as member function
8.         mInt += rhs.mInt;
9.         return *this;
10.     }
11.
12.private:
13.     int mInt{};
14.};
```

```
1. int main() {
2.     NewInt newInt1{6};
3.     NewInt newInt2{8};
4.     newInt1 += newInt2;
5.     std::cout << "The result is " <<
  newInt1.getInt() << ".\n";

6.     return 0;
7. }
```

UNSW SYDNEY

Friend Functions

- A ‘friend’ function is defined outside of the class, however, has full access to the class members of its arguments of the class
- Denoted with the “friend” keyword
- Generally, only used for specific operator overloading

Friend Functions – Continued

```cpp
#include <iostream>

class Cents {
public:
    Cents(int cents)
        : mCents{ cents } {
    }
    friend Cents operator+(const Cents &lhs, const Cents &rhs);
    friend std::ostream& operator<<(std::ostream &out, const Cents &cents);
private:
    int mCents {};
};

// note: this function is a friend function!
Cents operator+(const Cents &lhs, const Cents &rhs) {
    // use the Cents constructor and operator+(int, int)
    return Cents{ lhs.mCents + rhs.mCents };
}

std::ostream& operator<<(std::ostream &out, const Cents &cents) {
    out << cents.mCents;
    return out;
}
```

```cpp
int main() {
    Cents cents1{ 6 };
    Cents cents2{ 8 };
    Cents centsSum{ cents1 + cents2 };
    std::cout << "I have " << centsSum
  << " cents.\n";

    return 0;
}
```

STL

UNSW
SYDNEY

STL - Recap

- STL (standard template library) is part of the standard library and contains a set of customisable:
  - Containers
  - Iterators
  - Algorithms

We will go through each of these now

STL Containers

- STL containers can be created on any single or set of data types
- You have already seen these for Array and Vector so far
- Many other containers exist
- All containers have the same base (common interface) and they each extend this an implement functionality in their own way

STL Containers

- The following containers exist in the STL

Sequence containers	Container adaptors	Associative containers	Unordered Associative containers
array	stack	set	unordered_set
vector	queue	multiset	unordered_multiset
deque	priority_queue	map	unordered_map
forward_list		multimap	unordered_multimap
list			

STL Iterators

- Iterators are used to point to an individual element of a container
- Act similar to a raw pointers
- These allow for general code to be written that functions over all containers (sorting, searching, comparison)

Iterators *(from lecture)*

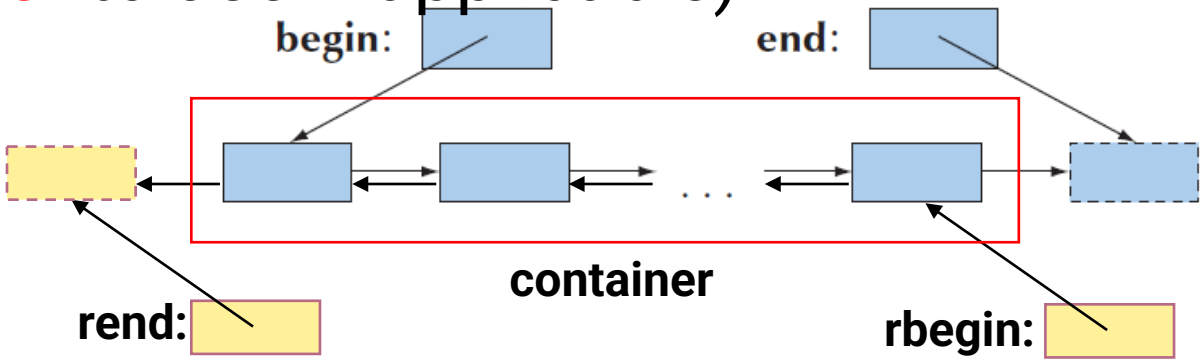
Four main types:

- .begin() – first element
- .end() – one after the final element
- .rbegin() – last element (for traversing in reverse order)
- .rend() – one before first element (in reverse order)

```
1. for (auto iter=v.begin(); iter!=v.end(); ++iter) {
2.   // your code
3. }
```

And the const versions, meaning the iterator will not allow modification to the container (safer to use if applicable)

- .cbegin() – const_iterator
- .cend() – const_iterator
- .crbegin() – const_reverse_iterator
- .crend() – const_reverse_iterator



Iterators Example

```
1.  #include <iostream>
2.  #include <vector>
3.  #include <iterator>

4.  // Use iterator to print the entire vector
5.  void print(const std::vector<int> &v) {
6.      for (std::vector<int>::const_iterator iter = v.cbegin(); iter != v.cend(); ++iter) {
7.      // for (auto iter = v.cbegin(); iter != v.cend(); ++iter) {
8.          std::cout << *iter << " ";   // dereference
9.      }
10.     std::cout << '\n';
11. }

12. void printReverse(const std::vector<int> &v) {
13.     for (std::vector<int>::const_reverse_iterator iter = v.crbegin(); iter != v.crend(); ++iter) {
14.     // for (auto iter = v.crbegin(); iter != v.crend(); ++iter) {
15.         std::cout << *iter << " ";   // dereference
16.     }
17.     std::cout << '\n';
18. }

19. int main() {
20.     std::vector<int> v{11, 55, 44, 33, 88};
21.     print(v);
22.     printReverse(v);
23.     return 0;
24. }
```

STL Algorithms

- The algorithm library in C++ is a collection of functions that act on a range of elements
  - A range is a sequence accessed by an iterator
- There are a wide variety of algorithms for different purposes
  - Review here: https://en.cppreference.com/w/cpp/algorithm
  - Remember when looking for algorithms, we are working with C++14 so don't select any that only support newer versions

Algorithm Parameters

- Most of the algorithms have one of the following four forms:
  - Where alg is the name of the algorithm and beg and end denote the input range on which the algorithm operates.

```
alg(beg, end, other args);
alg(beg, end, dest, other args);
alg(beg, end, beg2, other args);
alg(beg, end, beg2, end2, other args);
```

Algorithms Example

```
1.  int main() {
2.      // initialise from C-array using iterators
3.      int array[] {11, 55, 44, 33, 88, 99, 11, 22, 66, 77};
4.      std::vector<int> v(array, array + sizeof(array)/sizeof(int));
5.      print(v);
6.
7.      // Sort
8.      std::sort(v.begin(), v.end());  // entire container [begin(),end())
9.      print(v);
10.
11.     // Reverse
12.     std::reverse(v.begin(), v.begin() + v.size()/2);  // First half
13.     print(v);
14.
15.     // Random Shuffle
16.     std::random_shuffle(v.begin() + 1, v.end() - 1);  // exclude first and last elements
17.     print(v);
18.
19.     // Search
20.     int num {88};
21.     std::vector<int>::iterator found {find(v.begin(), v.end(), num)};
22.     // auto found {std::find(v.begin(), v.end(), num)};
23.     if (found != v.end()) {
24.         std::cout << "Found " << num << " at the " << std::distance(v.begin(), found) +1 << "th element.\n";
25.     }
26.     return 0;
27. }
```

UNSW SYDNEY

Lambda Functions

- Some algorithms take in a function that is called
- These are often passed as Lambda functions which can be passed directly as a parameter
  - This essentially defines a function in line
- They have the form:

$$[\textit{capture list}]\ (\textit{parameter list})\ \texttt{->}\ \textit{return type}\ \{\ \textit{function body}\ \}$$

  - where capture list is an (often empty) list of local variables defined in the enclosing function;
  - return type, parameter list, and function body are the same as in any ordinary function.

Lambda Functions Example

```
1.  #include <vector>
2.  #include <algorithm>
3.  #include <iostream>
4.
5.  // void print(const int& n) {
6.  //     std::cout << " " << n;
7.  // }
8.
9.  int main()
10. {
11.     std::vector<int> nums{3, 4, 2, 8, 15, 267};
12.
13.     auto print = [](const int& n) { std::cout << " " << n; };
14.
15.     std::cout << "before:";
16.     std::for_each(nums.cbegin(), nums.cend(), print);
17.     std::cout << '\n';
18.
19.     int i {1};
20.     std::for_each(nums.begin(), nums.end(), [i](int &n){ n+=i;});
21.
22.     std::cout << "after:";
23.     std::for_each(nums.cbegin(), nums.cend(), print);
24.     std::cout << '\n';
25. }
```

Lab tasks

UNSW
SYDNEY