ST. CHARLES COMMUNITY COLLEGE

CPT-182 - Programming in C++

Module 12

Sorting Algorithms, Exceptions

Dayu Wang

- **Sorting Algorithms**

→ A sorting algorithm can sort a vector in non-decreasing (increasing) order.

→ Some sorting algorithms use comparison to sort the vectors; others do not ~~use comparison~~.

→ In this class, you are required to understand four comparison sorting algorithms.

```
template<class T>
class Sorting {
public:
    // Static class-member functions
    void static selection_sort(vector<T>&);  // Selection sort
    void static bubble_sort(vector<T>&);  // Bubble sort
    void static insertion_sort(vector<T>&);  // Insertion sort
    void static merge_sort(vector<T>&);  // Merge sort
private:
    // Static class-member function

    // Merges two sorted vectors into a single sorted vector.
    void static merge(const vector<T>&, const vector<T>&, vector<T>&);
};
```

- **Selection Sort**

  → **In the first iteration, you put the minimum value in the vector in the first place; in the second iteration, you put the second minimum value in the vector in the second place; and so on.**

- **Demo of Selection Sort**

35	65	30	60	20	Swap 35 and 20.
20	65	30	60	35	Swap 65 and 30.
20	30	65	60	35	Swap 65 and 35.
20	30	35	60	65	No ~~swap~~
20	30	35	60	65	No ~~swap~~
20	30	35	60	65	Sorted

- **C++ Implementation of Selection Sort**

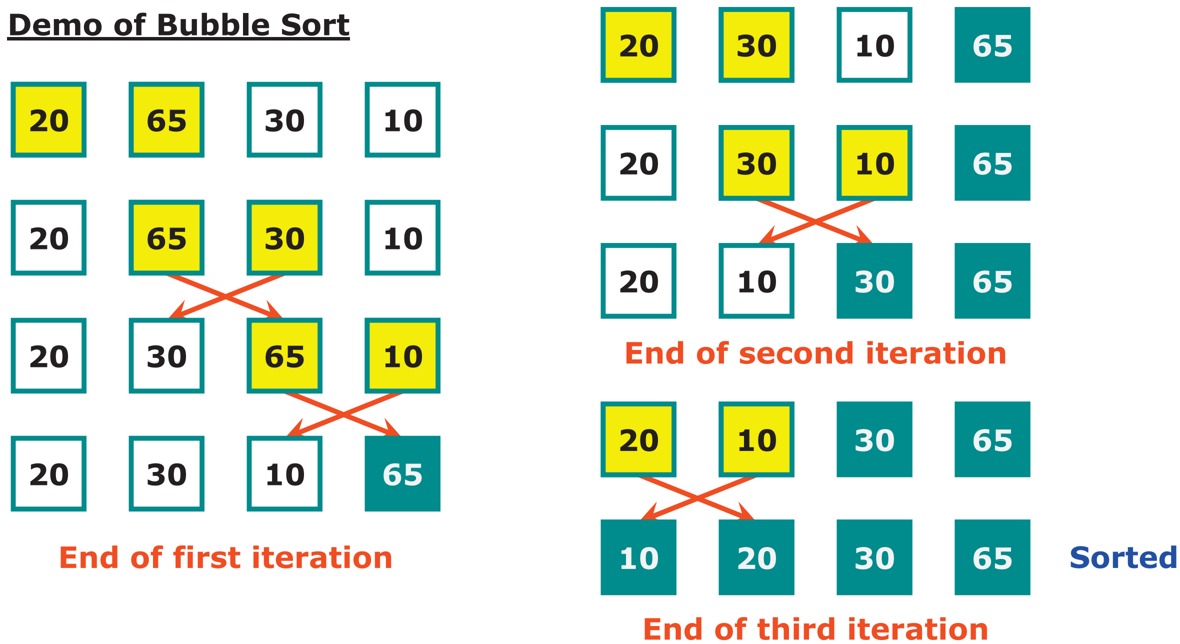
```cpp
// Selection sort
template<class T>
void Sorting<T>::selection_sort(vector<T>& vec) {
    for (size_t i = 0; i < vec.size(); i++) {
        // Stores the index of the min value in the rest of the vector.
        size_t min = i;

        // Find the min value in the rest of the vector.
        for (size_t j = i + 1; j < vec.size(); j++) {
            if (vec.at(j) < vec.at(min)) { min = j; }
        }

        // Swap vec[min] with vec[i] if they are not the same.
        if (min != i) { swap(vec.at(i), vec.at(min)); }
    }
}
```

- **Bubble Sort**

  → Compare the adjacent pairs (e.g., vec[0] and vec[1], vec[1] and vec[2], vec[2] and vec[3], and so on) of elements. If they are out of order, swap them.

  In the first iteration, you place the largest item; in the second iteration, you place the second largest item; and so on.

- **Demo of Bubble Sort**



- **C++ Implementation of Bubble Sort**

```
// Bubble sort
template<class T>
void Sorting<T>::bubble_sort(vector<T>& vec) {
    for (size_t i = 0; i < vec.size(); i++) {
        for (size_t j = 1; j < vec.size(); j++) {
            if (vec.at(j) < vec.at(j - 1)) {  // Out of order
                swap(vec.at(j - 1), vec.at(j));
            }
        }
    }
}
```

→ Can we improve this solution (code)?

- We do not need to compare values that are already placed (at the end of the vector).
- If in any iteration, no ~~swaps~~ were ever occurred, it means ________?

  It means that the vector is already sorted.

  Therefore, at the end of an iteration, if there were no ~~swaps~~ occurred in the iteration, then we can stop any further processing of the vector, since it is already sorted.

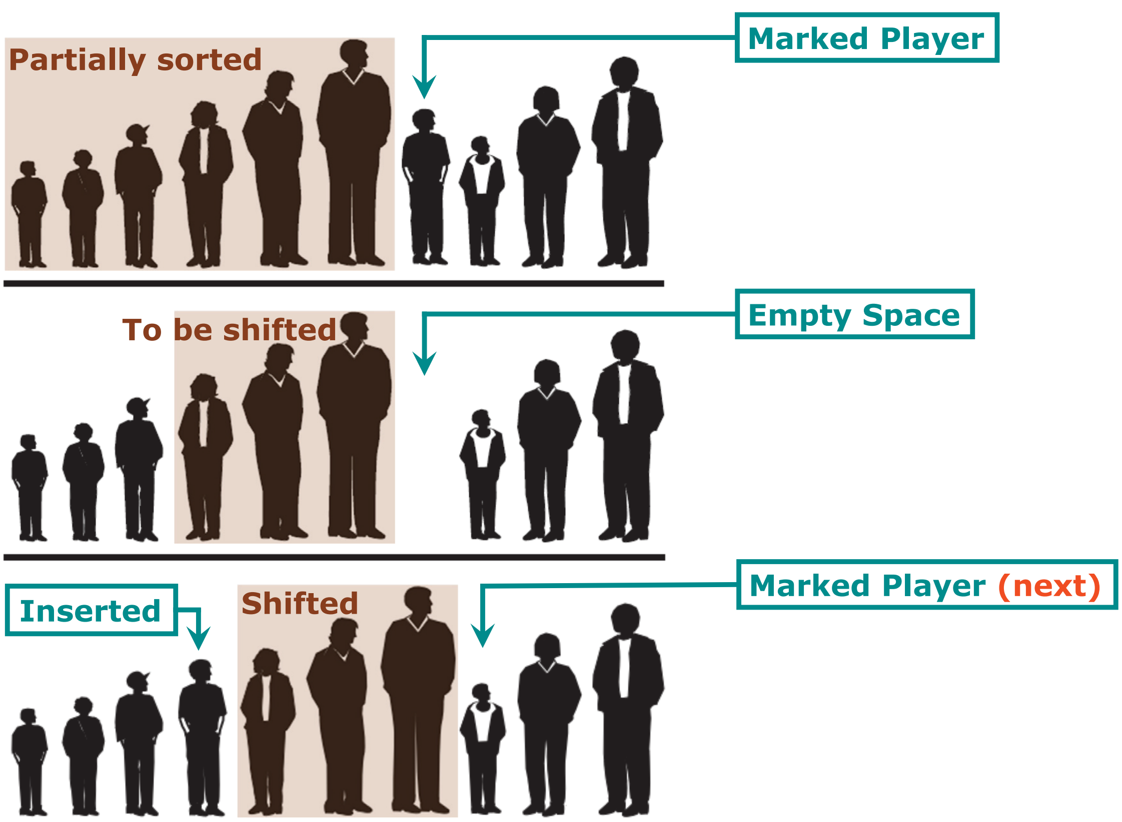
- **C++ Implementation of Bubble Sort (Improved)**

```
// Bubble sort
template<class T>
void Sorting<T>::bubble_sort(vector<T>& vec) {
    for (size_t i = 0; i < vec.size(); i++) {
        // Stores whether a swap occurs in this iteration.
        bool swapped = false;

        for (size_t j = 1; j < vec.size() - i; j++) {
            if (vec.at(j) < vec.at(j - 1)) {
                swap(vec.at(j - 1), vec.at(j));
                swapped = true;
            }
        }

        // If no swap occurred in this iteration,
        // then the vector is already sorted.
        if (!swapped) { return; }
    }
}
```

→ **In general, bubble sort is a bad sorting algorithm (slow).**

It uses lots of swaps. Swap is a slow operation.

- **Insertion Sort**



- **C++ Implementation of Insertion Sort**

```
// Insertion sort
template<class T>
void Sorting<T>::insertion_sort(vector<T>& vec) {
    for (size_t mark = 1; mark < vec.size(); mark++) {
        T key = vec.at(mark);
        int j;
        for (j = mark - 1; j >= 0 && vec.at(j) > key; j--) {
            vec.at(j + 1) = vec.at(j);
        }
        vec.at(j + 1) = key;
    }
}
```

→ **Insertion sort uses data shifts instead of ~~data swaps~~.**

Data shifts are faster than data swaps.

- **Merge Sort**

→ **Merge sort contains two parts:**

1) **Merge operation**
2) **Merge sort algorithm**

- **Merge Operation**

→ **How to merge two already sorted vectors into a single sorted vector?**

	i	i	i	i	i
Array 1:	23	47	81	95	

Output:	7	14	23	39	47	55	62	74	81	95

	j	j	j	j	j	j	j
Array 2:	7	14	39	55	62	74	

- **C++ Implementation of the Merge Operation**

```cpp
/** Merges two sorted vectors into a single sorted vector.
    @param vec_1: first sorted vector to merge
    @param vec_2: second sorted vector to merge
    @param out: single output sorted vector
*/
template<class T>
void Sorting<T>::merge(const vector<T>& vec_1, const vector<T>& vec_2,
                       vector<T>& out) {
    size_t i = 0, j = 0, k = 0;
    while (i < vec_1.size() && j < vec_2.size()) {
        if (vec_1.at(i) <= vec_2.at(j)) { out.at(k++) = vec_1.at(i++); }
        else { out.at(k++) = vec_2.at(j++); }
    }
    while (i < vec_1.size()) { out.at(k++) = vec_1.at(i++); }
    while (j < vec_2.size()) { out.at(k++) = vec_2.at(j++); }
}
```


- **Merge Sort**

→ **Basic Idea (Algorithm):**

1) Divide the entire vector into two halves, the left half and right half.
2) Sort the left half.
3) Sort the right half.
4) Merge the sorted left half and right half to form sorted whole vector.

→ **[Important] How to "sort the left half"? How to "sort the right half"?**

Merge sort is a recursive algorithm.

→ **Algorithm**

1) If the size of the vector is less than 2, then return (base case).
2) Copy the left half of the vector into another vector (denoted as `left_half`).
3) Copy the right half of the vector into another vector (`right_half`).
4) Recursively sort `left_half`.
5) Recursively sort `right_half`.
6) Merge `left_half` and `right_half`.

- **C++ Implementation of Merge Sort**

```cpp
// Merge sort
template<class T>
void Sorting<T>::merge_sort(vector<T>& vec) {
    // Base case
    if (vec.size() < 2) { return; }

    // Copy the left half of the vector into another vector.
    vector<T> left_half(vec.size() / 2);
    copy(vec.begin(), vec.begin() + vec.size() / 2, left_half.begin());

    // Copy the right half of the vector into another vector.
    vector<T> right_half(vec.size() - left_half.size());
    copy(vec.begin() + vec.size() / 2, vec.end(), right_half.begin());

    // Sort "left_half" and "right_half" recursively.
    merge_sort(left_half);
    merge_sort(right_half);

    // Merge the sorted left half and right half.
    merge(left_half, right_half, vec);
}
```


- **Exceptions**

→ **An exception is a circumstance that a program was not designed to handle, e.g., user enters a negative height.**

→ **Why we need exceptions?**

- **We cannot assume that the user will always provide valid input.**
- **We do not want the program crash due to invalid input provided by the user.**
- **If the input of the program is invalid, we would like our program to terminate gracefully, instead of a ~~crash~~.**

→ **[Example] Index can be out of bounds.**

```cpp
int main() {
    vector<int> vec = { 11, 13, 15, 17 };
    cout << "Enter an index: ";
    int index;
    cin >> index;
    cout << "vec[" << index << "] = " << vec.at(index) << endl;
    system("pause");
    return 0;
}
```

What if the use enters an index that is out of bounds?

When you see red cross, that means your program crashed.

How to gracefully tell the user that the input index is out of bounds?

```
int main() {
    vector<int> vec = { 11, 13, 15, 17 };
    cout << "Enter an index: ";
    int index;
    try {
        cin >> index;
        cout << "vec[" << index << "] = " << vec.at(index) << endl;
    } catch (exception e) {
        cout << "[Invalid Input] Index out of bounds" << endl;
    }
    system("pause");
    return 0;
}
```


- <u>`try-catch` **Blocks**</u>

→ **A `try` block surrounds normal code, which is exited immediately if a `throw` statement executes.**

→ **A `catch` clause immediately follows a `try` block; if the catch was reached due to an exception thrown of the catch clause's parameter type, the clause executes. The clause is said to catch the thrown exception.**

A `catch` block is called a handler because it handles an exception.

- **throw Statement**

→ **A `throw` statement appears within a `try` block; if reached, execution jumps immediately to the end of the `try` block.**

The code is written so only error situations lead to reaching a `throw`.

The `throw` statement can throw anything, such as an object of type `exception` or its subclasses.

[Not-To-Do] In C++, technically, you can throw an `int` or `string`, but it is not good programming habit. Please always throw exceptions (or its subclasses).

The statement is said to throw an exception of the particular type.

A `throw` statement's syntax is similar to a `return` statement.

→ **[Example] Remove the last element from a vector.**

```
template<class T>
void remove_last(vector<T>& vec) {
    if (vec.empty()) {
        throw exception("Accessing empty vector");
    }
    vec.pop_back();
}
```



```
int main() {
    vector<int> vec = { 11, 13, 15, 17 };
    cout << "Enter an index: ";
    int index;
    try {
        cin >> index;
        cout << "vec[" << index << "] = " << vec.at(index) << endl;
        vec.clear();
        remove_last(vec);
    } catch (out_of_range e) {
        cout << "[Invalid Input] Index out of bounds" << endl;
    } catch (exception e) {
        cout << e.what() << endl;
    }
    system("pause");
    return 0;
}
```

→ **The `.what()` function returns the "error message" of the exception.**