UNIVERSITY OF WATERLOO
FACULTY OF ENGINEERING
Department of Electrical &
Computer Engineering

ECE 150 *Fundamentals of Programming*

# Character sets

Douglas Wilhelm Harder, M.Math., LEL
Prof. Hiren Patel, Ph.D., P.Eng.
Prof. Werner Dietl, Ph.D.

## Outline

- In this lesson, we will:
  - Describe a character set
  - Look at four different implementations

## Character sets

- A *set* is a collection of unique items
  - There is no duplication of entries

- A set of characters is a set where the entries are characters

## `unsigned char`

- A char in C++ is just a one-byte integer where the different values are interpreted as characters when printed
  - For example, the character `'*'` has the value 42 or `0x2A` or `0b00101010`

- The ASCII character set only uses 0 through 127
  - These can be stored as either
    - `unsigned char` 0, …, 255
    - `signed char` -128, …, 127
- Recall that int is signed, so you could use:

  ```
  signed int n{ 0 };
  ```
  - This is, however, usually unnecessary and redundant
- The C++ standard, however, does not specify if char is signed or unsigned
  - Thus, if you using more than 0, …, 127, you should specify this

## Character sets

- We could represent a set as characters:

  `{'C','F','i',',','d','S','e','9','!'}`

- Unfortunately, some characters are not printable,

  so we will revert to using integers 0, ..., 255

  {67, 70, 105, 44, 100, 83, 101, 57, 33}

## Operations on sets

- Operations we can perform on a set include:
  - Asking how many items are in the set
  - Adding a new member into the set if it isn't already in the set
  - Removing a member of a set
  - Emptying a set

## Operations on sets

- If you have two sets, you can
  - Take the *union* of the two sets:
    - This includes members that are in one set or the other

      {1, 3, 4, 5, 9, 10} ∪ {2, 3, 5, 6, 10} = {1, 2, 3, 4, 5, 6, 9, 10}
  - Take the *intersection* of two sets:
    - This includes all members that are in one set and the other

      {1, 3, 4, 5, 9, 10} ∩ {2, 3, 5, 6, 7, 10} = {3, 5, 10}
  - Subtract one set from another
    - Remove from the first set any entries that appear in the second

      {1, 3, 4, 5, 9, 10} – {2, 3, 5, 6, 10} = {1, 4, 9}

      {2, 3, 5, 6, 10} – {1, 3, 4, 5, 9, 10} = {2, 6}

## Operations on sets

- We will implement these three as member functions:

```
set1.set_union( set2 );
set1.set_intersection( set2 );
set1.set_minus( set2 );
```

  – Like automatic assignment operators (e.g., `+=`, `*=`)
    - For example, after executing

```
set1.set_union( set2 );
```

      `set1` becomes the union of what was `set1` and `set2`

## Possible designs

- We will look at four implementations and discuss a fifth
  - An array of those characters in the set
  - A sorted array of those characters in a set
  - An array of 256 bool, with `true`/`false` for each character
  - An array of 256 bits, with `0`/`1` for each character
  - The fifth will be described in your course on algorithms and data structures

## Design 1

- We will start with the simplest design:
  - We will store the characters in the set in an array

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 'C' | 'F' | 'i' | ',' | 'd' | 'S' | 'e' | '9' | '!' | |

  - When we insert a new character, it gets appended to the end

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 'C' | 'F' | 'i' | ',' | 'd' | 'S' | 'e' | '9' | '!' | '8' |

    - If the array is full, we'll double the capacity
  - When we remove a character, we replace it with the last character

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 'C' | 'F' | 'i' | ',' | 'd' | 'S' | 'e' | '9' | '!' | '8' |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 'C' | 'F' | 'i' | ',' | 'd' | '8' | 'e' | '9' | '!' | |

## Design 1

- The set union, intersection and minus require us to step through both arrays:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 'C' | 'F' | 'i' | ',' | 'd' | 'S' | 'e' | '9' | '!' | |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| '8' | 'A' | 'n' | 'm' | 's' | 'e' | 'S' | '.' | | |

  - The union requires more memory

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 'C' | 'F' | 'i' | ',' | 'd' | 'S' | 'e' | '9' | '!' | '8' | 'A' | 'n' | 'm' | 's' | '.' | | | | | |

  - The intersection contains only two characters:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 'S' | 'e' | | | | | | | | |

  - Subtracting the second set from the first yields:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 'C' | 'F' | 'i' | ',' | 'd' | '9' | '!' | | | |

## Design 1

- Design issues:
  - You must check each entry in the set before determining a character is not in the set
    - Okay if there are only a dozen characters
    - Relatively expensive if there are a hundred or more…
  - Printing the entries in order is also expensive
  - The union, intersection and set minus operations are also potentially expensive
    - For the union, we must check if each entry is in this set,
      and if not, add it

## Design 2

- Next, we continue by using a sorted array:
  - The characters are sorted by their ASCII value

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| '!' | ',' | '9' | 'C' | 'F' | 'S' | 'd' | 'e' | 'i' | | |

  - When we insert a new character, the order specifies its position

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| '!' | ',' | '8' | '9' | 'C' | 'F' | 'S' | 'd' | 'e' | 'i' | |

    - If the array is full, we'll double the capacity
  - When we remove a character, characters get shifted to the left

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| '!' | ',' | '8' | '9' | 'C' | 'F' | 'S' | 'd' | 'e' | 'i' | |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| '!' | ',' | '8' | '9' | 'C' | 'F' | 'd' | 'e' | 'i' | | |

## Design 2

- We can speed up the set operations

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| '!' | ',' | '9' | 'C' | 'F' | 'S' | 'd' | 'e' | 'i' | |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| '.' | '8' | 'A' | 'S' | 'e' | 'm' | 'n' | 's' | | |

  - We can merge the two sorted arrays

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| '!' | ',' | '.' | '8' | '9' | 'A' | 'C' | 'F' | 'S' | 'd' | 'e' | 'i' | 'm' | 'n' | 's' | | | | | |

  - The intersection can also be performed with a merge:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 'S' | 'e' | | | | | | | | |

  - Similarly, subtracting the second from the first is a merge

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| '!' | ',' | '9' | 'C' | 'F' | 'd' | 'i' | | | |

## Design 2

- Design issues:
  - Determining membership can be with a binary search
  - Inserting or removing entries can be expensive
  - The union, intersection and difference operators are much faster

## Design 3

`{'C', 'F', 'i', ',', 'd', 'S', 'e', '9', '!'}`

- Next, we have an array of 256 Boolean values

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| F | F | F | F | F | F | F | F | F | F | F | F | F | F | F | F |
| F | F | F | F | F | F | F | F | F | F | F | F | F | F | F | F |
| F | T | F | F | F | F | F | F | F | F | F | F | T | F | F | F |
| F | F | F | F | F | F | F | F | F | T | F | F | F | F | F | F |
| F | F | F | T | F | F | T | F | F | F | F | F | F | F | F | F |
| F | F | F | T | F | F | F | F | F | F | F | F | F | F | F | F |
| F | F | F | F | T | T | F | F | F | T | F | F | F | F | F | F |
| F | F | F | F | F | F | F | F | F | F | F | F | F | F | F | F |
| F | F | F | F | F | F | F | F | F | F | F | F | F | F | F | F |
| F | F | F | F | F | F | F | F | F | F | F | F | F | F | F | F |
| F | F | F | F | F | F | F | F | F | F | F | F | F | F | F | F |
| F | F | F | F | F | F | F | F | F | F | F | F | F | F | F | F |
| F | F | F | F | F | F | F | F | F | F | F | F | F | F | F | F |
| F | F | F | F | F | F | F | F | F | F | F | F | F | F | F | F |
| F | F | F | F | F | F | F | F | F | F | F | F | F | F | F | F |
| F | F | F | F | F | F | F | F | F | F | F | F | F | F | F | F |

## Design 3

- Inserting or removing entries simply has one change the corresponding entry

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| F | F | F | F | F | F | F | F | F | F | F | F | F | F | F | F |
| F | F | F | F | F | F | F | F | F | F | F | F | F | F | F | F |
| F | T | F | F | F | F | F | F | F | F | F | F | T | F | F | F |
| F | F | F | F | F | F | F | F | F | T | F | F | F | F | F | F |
| F | F | F | T | F | F | T | F | F | F | F | F | F | F | F | F |
| F | F | F | T | F | F | F | F | F | F | F | F | F | F | F | F |
| F | F | F | F | T | T | F | F | F | T | F | F | F | F | F | F |
| F | F | F | F | F | F | F | F | F | F | F | F | F | F | F | F |
| F | F | F | F | F | F | F | F | F | F | F | F | F | F | F | F |
| F | F | F | F | F | F | F | F | F | F | F | F | F | F | F | F |
| F | F | F | F | F | F | F | F | F | F | F | F | F | F | F | F |
| F | F | F | F | F | F | F | F | F | F | F | F | F | F | F | F |
| F | F | F | F | F | F | F | F | F | F | F | F | F | F | F | F |
| F | F | F | F | F | F | F | F | F | F | F | F | F | F | F | F |
| F | F | F | F | F | F | F | F | F | F | F | F | F | F | F | F |
| F | F | F | F | F | F | F | F | F | F | F | F | F | F | F | F |

## Design 3

- Design issues:
  - Very fast determination of membership
  - Slow to calculate unions, intersections and differences
  - Requires an array of 256 entries, even with only 3 members

## Design 3

- As for the operations:
  - An entry is in the union if it is in either set:
    - For each of the 256 indices,
      if `array[k] == false` and `other_set.array[k] == true`,
      set `array[k] = true;`
  - An entry is in the intersection if it is in both sets:
    - For each of the 256 indices,
      if `array[k] == true` and `other_set.array[k] == false`,
      set `array[k] = false;`
  - An entry is in the difference if it is in the set and not in the other:
    - For each of the 256 indices,
      if `array[k] == true` and `other_set.array[k] == true`,
      set `array[k] = false;`

## Design 4

`{'C', 'F', 'i', ',', 'd', 'S', 'e', '9', '!'}`

- Finally, store and array of eight (8) unsigned integers (32 bits each), for a total of 256 bits
  - `0` means it is not in the set, `1` means it is in the set

```
00000000000000000000000000000000
01000000000010000000000001000000
00010010000000000001000000000000
00001100010000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
```

## Design 4

`{'C', 'F', 'i', ',', 'd', 'S', 'e', '9', '!'}`

- The first three bits of the unsigned char indicate which of the eight integers it is stored in
- The last five bits indicate which of the 32 bits it is stored in

```
array[char >> 5] & (1 << (ch & 0b1111))
```

```
00000000000000000000000000000000
01000000000010000000000001000000
00010010000000000001000000000000
00001100010000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
```

## Design 4

- As for the operations, we only have to perform the appropriate bitwise operations on each of the eight integers:
  - Union:
    ```
    array[k] |= other_set.array[k];
    ```
  - Intersection:
    ```
    array[k] &= other_set.array[k];
    ```
  - Difference:
    ```
    array[k] &= ~other_set.array[k];
    ```

## Design 4

- Design issues:
  - Almost-as-fast membership
  - One eighth the memory of Design 3
  - Very fast bitwise operations

- Given a choice between Design 3 and Design 4,
  Design 4 is superior, just more difficult to work with…

## Comparison

- Memory use

| Design | Few entries | Unknown | Many entries |
|---|---|---|---|
| 1 | Good | Poor | Very poor |
| 2 | Good | Poor | Very poor |
| 3 | Very poor | Poor | Okay |
| 4 | Poor | Okay | Good |

- Determining membership

| Design | Few entries | Unknown | Many entries |
|---|---|---|---|
| 1 | Fast | Slow | Slow |
| 2 | Okay | Okay | Okay |
| 3 | Fast | Fast | Fast |
| 4 | Fast | Fast | Fast |

## Comparison

- Inserting or removing entries

| Design | Few entries | Unknown | Many entries |
|---|---|---|---|
| 1 | Fast | Slow | Slow |
| 2 | Fast | Slow | Slow |
| 3 | Fast | Fast | Fast |
| 4 | Fast | Fast | Fast |

- Set operations: union, intersection and set minus

| Design | Few entries | Unknown | Many entries |
|---|---|---|---|
| 1 | Fast | Slow | Very slow |
| 2 | Fast | Okay | Okay |
| 3 | Slow | Okay | Fast |
| 4 | Slow | Okay | Fast |

## Conclusions

- While you may think that one of these designs should be “best,” this is certainly not the case
  - If you expect few entries, choose Design 1
  - If you expect on average many entries, choose Design 4
  - If you don’t know how many entries there will be, choose Design 3
  - Design 4 is a more efficient bitwise version of Design 3, so there is no real benefit of Design 3

## Design 5

- In your course on algorithms and data structures, you will be introduced to balanced search trees
  - These are more efficient on average than sorted arrays, but also use more memory

- The `std::set` class in the standard library uses such trees

## Trying these out

- We provide implementations of each of these four designs
  - At the very least, consider what member functions you'd use, and how would you author those algorithms
  - Looking at the solutions immediately is not going to help…

## Trying these out

```
// Class declarations
class Character_set;

// Class defintions
class Character_set {
    public:
        Character_set();

        bool member( unsigned char ch ) const;
        std::size_t size() const;
        void print() const;

        bool insert( unsigned char ch );
        bool remove( unsigned char ch );
        void clear();

        Character_set &set_union( Character_set const &S );
        Character_set &set_intersection( Character_set const &S );
        Character_set &set_minus( Character_set const &S );
        bool operator==( Character_set const &S ) const;
        bool operator!=( Character_set const &S ) const;

    private:
        // Enter your member variables and any private
        // member functions here...
};
```

## Trying these out

- This function only depends on `member(…)` working correctly:

```
void Character_set::print() const {
    std::cout << "{";

    bool is_first_printed{ false };

    for ( std::size_t k{ 0 }; k < 256; ++k ) {
        if ( member( k ) ) {
            if ( is_first_printed ) {
                std::cout << ", ";
            } else {
                is_first_printed = true;
            }

            std::cout << k;
        }
    }

    std::cout << "}" << std::endl;
}
```

## Trying these out

- Implement the member functions one by one

```
Character_set::Character_set() {
  // Implement your constructor here
}

bool Character_set::member( unsigned char ch ) const {
  return false;
}

std::size_t Character_set::size() const {
  return 0;
}

bool Character_set::insert( unsigned char ch ) {
  return false;
}

bool Character_set::remove( unsigned char ch ) {
  return false;
}

void Character_set::clear() {
}
```

```
Character_set &Character_set::set_union(
  Character_set const &S
) {
  return *this;
}

Character_set &Character_set::set_intersection(
  Character_set const &S
) {
  return *this;
}

Character_set &Character_set::set_minus(
  Character_set const &S
) {
  return *this;
}

bool Character_set::operator==( Character_set const &S ) const{
  return false;
}

bool Character_set::operator!=( Character_set const &S ) const {
  return true;
}
```

## Summary

- Following this lesson, you now:
  - Understand the idea of a set of characters
    - A set that has a maximum number of 256 possible entries
  - Have seen four designs
  - Considered four different approaches
    - The fourth is an improvement on the third…
  - Considered strengths and weaknesses of each
  - Have hopefully observed there is never an ideal design
  - Hopefully are motivated to implement some of these designs

## References

[1] https://en.cppreference.com/w/cpp/container/set

## Colophon

These slides were prepared using the Georgia typeface. Mathematical equations use Times New Roman, and source code is presented using `Consolas`.

The photographs of lilacs in bloom appearing on the title slide and accenting the top of each other slide were taken at the Royal Botanical Gardens on May 27, 2018 by Douglas Wilhelm Harder. Please see

https://www.rbg.ca/

for more information.

## Disclaimer