# COMP 1633: Intro to CS II

# ADT Case Study

Charlotte Curtis

March 25, 2024

# Where we left off

- `const` correctness with classes
- Constructors and destructors
- Function and operator overloading

> *Textbook Sections 10.2, 11.2*

```cpp
class Time {
public:
    Time(int h, int m, int s);
    Time();

    void write(std::ostream &out) const;
    void increment();
    bool operator<(const Time &t) const;
private:
    int hours;
    int minutes;
    int seconds;

    int compare(const Time &t) const;
};
```

# Today's topics

- More overloading: stream operators
- A common abstract data type: stacks

> *Textbook Sections 11.2, 13.2*

# Rules of operator overloading

- Built-in operators cannot be redefined
  - e.g. can't redefine `.` or `::`
- Only existing operators can be overloaded
  - e.g. can't define an `@` operator
- Precedence and associativity are the same as for the built-in operators
  - e.g. `+` has higher precedence than `==`
- Cannot change the number of arguments that an operator takes
  - e.g. `+` can't be redefined to take 3 arguments

# Conventions of operator overloading

In general, the purpose of operator overloading is to make code **more readable**

- Keep the **semantics** of the operator the same
  - Don't redefine `+` to mean subtraction!
- Provide the operator only if its meaning is obvious
  - `Time + Time` is obvious, but what about `Time * Time`?
- If one operator is overloaded, all related operators should be overloaded
  - if you overload `<`, you should also overload `>`, `<=`, `>=`, and `==`
  - If you overload `+`, you should probably also provide `-`, `+=`, and `-=`

# Overloading thus far

- We can overload **functions** (like constructors)

```
bool is_yummy(const char *food);
bool is_yummy(const std::string &food);
```

- We can overload **binary operators** (like `==` or `[]`)

```
bool operator[](int i) const; // access element i in a list-type ADT
```

- We haven't yet done **stream operators** (`<<` and `>>`)
- We also skipped over **unary operators** (`++` and `!`)

# Member vs non-member overloading

- While I've introduced overloading operators in the context of classes, this is **not necessary** - operators can be overloaded as non-member functions
- For example, we *could* have defined `operator==` as a non-member function:

```
bool operator==(const Time &lhs, const Time &rhs) {
    return (lhs.hours == rhs.hours
            && lhs.minutes == rhs.minutes
            && lhs.seconds == rhs.seconds);
}
```

- However, we made the member variables `private`, so this doesn't work
- Also keep in mind, **member functions** pass the **left-hand side** as `this`

# Overloading stream operators

- Taking a look at the documentation for `operator<<`, you can see that it's an **overloaded member function** of `std::ostream`
- It's also clear when you use it that the **left-hand side** is the stream, and the **right-hand side** is the thing you're printing

```
cout << "Hello, world!" << endl;
```

- This is a problem! We can't overload `operator<<` in the `Time` class
- It needs to be a **non-member function** that takes a `std::ostream` as the LHS
- That's a problem too though - we can't access the `private` member variables

# Possible solution: `friend` functions

- We can make the operator overload a `friend` of the `Time` class
- `friend` functions are declared as part of the class declaration (usually the `public` section), but they are **not** member functions
- To quote the textbook: *"Friends can access private members"*

> *Note: this is somewhat contentious, as `friend`s kind of break encapsulation. Can you think of a way to implement this without `friend`?*

# Overloading stream operators

- `friend` function declaration:

```
class Time {
public:
    friend std::ostream &operator<<(std::ostream &out, const Time &t);
... etc
```

- The implementation doesn't use the keyword `friend`, but it can magically access the private members::

```
std::ostream &operator<<(std::ostream &out, const Time &t) {
    out << t.hours << ':' << t.minutes << ':' << t.seconds;
    return out;
}
```

# Operator overloading check-in 1/2

The **primary purpose** of operator overloading is to:

A. Improve memory efficiency

B. Improve performance

C. Improve readability

D. Make classes easier to write

E. Encapsulation

# Operator overloading check-in 2/2

The `<<` operator can't be overloaded as a member function because:

A. The left-hand side is a `std::ostream`

B. It's a binary operator

C. It needs to be a `const` function

D. It needs to be `public`

E. It wants to have a `friend`

# A common ADT: Stacks

- What makes a class an **abstract data type**?
    - It has a valid **domain** (set of values)
    - It has **operations** that can be performed on it
    - It **hides** the implementation details
- Our `Time` class is a (simple) ADT, but it's pretty boring
- Let's look at a more interesting one: **stacks**

# Stacks

- Just like it sounds, a stack is a data storage structure that lets you:
  - put stuff on the top of the stack
  - take stuff off the top of the stack
- This is called **LIFO** (last in, first out)
- In computer science, stacks are used for:
  - the **function call stack**, aka "the stack"
  - **undo** operations in most programs
  - bash command history (up arrow)
  - The "back" button in your browser

# Specifying the ADT

- We need to specify the **domain** and **operations** for our stack ADT
- **Domain**:
  - a homogenous base type, like `int` or `std::string`
  - grows and shrinks dynamically, some reasonable max capacity
- **Operations**:
  - create an empty stack
  - check if the stack is empty/full (`empty/full`)
  - add/remove an element to the "top" of the stack (`push/pop`)
  - Look at the top element without removing (`peek`)

# Sample interface

```
class StringStack {
public:
    StringStack(int capacity);
    ~StringStack();

    bool empty() const;
    bool full() const;
    void push(const std::string &s);
    std::string pop();
    std::string peek() const;
private:
    ???
};
```

# Sample usage

Let's implement browser history using our stack ADT:

```
StringStack history(10); // max 10 pages
history.push("https://www.mymru.ca/");
history.push("https://stackoverflow.com/");
history.push("https://www.funnycatvideos.com/");

// go back to the previous page
load_url(history.pop());

// hover over the back button
if (history.empty())
    show_message("First page, can't go back\n");
else:
    show_message("Click to go back to: " + history.peek());
```

# `StringStack` implementation V1

- Based on its usage and public interface, how is `StringStack` implemented?
- Option 1: Arrays

```
class StringStack {
public:
...
private:
    int capacity;
    std::string *stack;
    ???
}
```

- Problem: remember how arrays need **shifting** to add to the "head"?
- Solution: who cares which end is the head!

# Complete `private` section for V1

```cpp
class StringStack {
public:
...
private:
    int capacity;
    std::string *stack; // pointer to the array
    int top; // index of the top element
}
```

- `top` is the index of the top element
- What should `top` be if the stack is empty?

# `StringStack` implementation V2

- Adding/removing elements at the head is easy for **linked lists**
- Option 2: Linked list

```
class StringStack {
public:
...
private:
    struct Node {
        std::string data;
        Node *next;
    };
    Node *head; // pointer to the head node
    ???
```

- Problem: there's no inherent **capacity** for a linked list
- Solution: add a counter to keep track of number of elements

# Complete `private` section for V2

```cpp
class StringStack {
public:
...
private:
    struct Node {
        std::string data;
        Node *next;
    };
    Node *head; // pointer to the head node
    int capacity;
    int size; // number of elements in the stack
}
```

- The `Node` struct is `private` because it's an implementation detail

# Constructors/destructors

Note: `using namespace std;` shouldn't go in the header file, but it's okay in `.cpp`

```cpp
// Array version (V1)
StringStack::StringStack(int capacity) {
    string *stack = new string[capacity];
    this->capacity = capacity;
    top = -1;
}

StringStack::~StringStack() {
    delete[] stack;
}
```

```cpp
// Linked list version (V2)
StringStack::StringStack(int capacity) {
    head = NULL;
    this->capacity = capacity;
    size = 0;
}

StringStack::~StringStack() {
    Node *curr = head;
    while (curr) {
        Node *next = curr->next;
        delete curr;
        curr = next;
    }
}
```

# `empty`, `full`, and `peek`

```cpp
// Array version (V1)
bool StringStack::empty() const {
    return top == -1;
}

bool StringStack::full() const {
    return top == capacity - 1;
}

std::string StringStack::peek() const {
    if (empty())
        return "";
    return stack[top];
}
```

```cpp
// Linked list version (V2)
bool StringStack::empty() const {
    return size == 0;
}

bool StringStack::full() const {
    return size == capacity;
}

std::string StringStack::peek() const {
    if (empty())
        return "";
    return head->data;
}
```

# push and pop

```cpp
// Array version (V1)
void StringStack::push(const std::string &s) {
    if (full())
        return;
    stack[++top] = s;
}

std::string StringStack::pop() {
    if (empty())
        return "";
    return stack[top--];
}
```

```cpp
// Linked list version (V2)
void StringStack::push(const std::string &s) {
    if (full())
        return;
    Node *new_node = new Node;
    new_node->data = s;
    new_node->next = head;
    head = new_node;
    size++;
}

std::string StringStack::pop() {
    if (empty())
        return "";
    std::string data = head->data;
    Node *next = head->next;
    delete head;
    head = next;
    size--;
    return data;
}
```

# Summary

- The linked list implementation is more complex, but with one big advantage: **no max capacity**
- In fact, keeping track of the size adds to the complexity
- The array implementation *could* dynamically resize whenever you try to push to a full stack, but this is also introducing complexity
- Which one is better? Depends on your use case!

# Coming up Next

- **Assignment 4** 🎉 - refactoring Assignment 3 to use an ADT
- Lab Exercise: Designing an ADT
- Next lecture: Something totally different: recursion!

> *Textbook Chapter 14*