Intro to C++ Classes

CMPT 125
Mo Chen
SFU Computing Science
9/3/2020
Lecture 23

Today

● Why not `struct`? Why `class`?
● C++ Encapsulation
● C++ Information Hiding
Marrying Data and Functions (Review)

Encapsulation

- bundle related data and operations together

Forge a language construct that joins data and operations together

- use a struct! use a class! (C++)
- make the functions part of the data type explicitly
  - called *methods*
- similar idea to an *object* in Python

Adds another level of protection against misuse
typedef struct _queue {
    LL_t * intlist;
} queue_t;

queue_t * queue_create(void);
void queue_destroy(queue_t * q);
int queue_isEmpty(queue_t * q);
void queue_enqueue(queue_t * q, int element);
int queue_dequeue(queue_t * q);

typedef struct _queue {
    LL_t * intlist;
    void queue_destroy(queue_t * q);
    int queue_isEmpty(queue_t * q);
    void queue_enqueue(queue_t * q, int element);
    int queue_dequeue(queue_t * q);
} queue_t;

queue_t * queue_create(void);
typedef struct _queue {
    LL_t * intlist;
} queue_t;

queue_t * queue_create(void);

void queue_destroy(queue_t * q);
int queue_isEmpty(queue_t * q);
void queue_enqueue(queue_t * q, int element);
int queue_dequeue(queue_t * q);
typedef struct _queue {
    LL_t * intlist;
} queue_t;

queue_t * queue_create(void);
void queue_destroy(queue_t * q);
int queue_isEmpty(queue_t * q);
void queue_enqueue(queue_t * q, int element);
int queue_dequeue(queue_t * q);

typedef struct _queue {
    LL_t * intlist;
    void (* destroy)(struct _queue * q);
    int (* isEmpty)(struct _queue * q);
    void (* enqueue)(struct _queue * q, int element);
    int (* dequeue)(struct _queue * q);
} queue_t;

queue_t * queue_create(void);

Pointer to a function rather than the function itself

Caller's notation:
Q->enqueue(Q, x);
A Look Ahead to C++

Motivated by these interface issues, C++ evolved out of C.

- formulated by Bjarne Stroustrup in 1978

Provides the syntactic sugar for:

- information hiding
- encapsulation of data and methods
- common code re-use situations

Migrate from `struct` → `class`
Step 1: The `class / public` Keywords

Instead of:    `typedef struct { ... } queue_t;`
Use:            `class queue { ... };`

Adjust types from `queue_t` → `queue`

Add the `public:` keyword (just until we get to Step 4)
Step 2: Add The Methods

Migrate the functions into the class definition → methods

- except for `_create()` and `_destroy()` which are special cases

Remove the `queue_` prefix from method names
Step 2: Add The Methods

Migrate the functions into the class definition → methods
- except for _create() and _destroy() which are special cases

Remove the queue_prefix from method names

Remove parameter queue * q
Step 2: Add The Methods

Migrate the functions into the class definition → methods

- except for _create() and _destroy() which are special cases

Remove the queue_ prefix from method names

Remove parameter queue * q

- every method has direct access to all fields

Methods can use the C++ keyword this if they need such a pointer
Step 3: The class:: Syntax

class queue {
    public:
        LL_t * intlist;
        int isEmpty();
        void enqueue(int data);
        int dequeue();
};
queue * queue_create(void);
void queue_destroy(queue * q);

(int header file queue.h)

(int queue_isEmpty(queue_t * q) {
    return (q->intlist->head == NULL);
}

void queue_enqueue(queue_t * q, int data) {
    LLappend(q->intlist, data);
}

int queue_dequeue(queue_t * q) {
    return LLremoveHead(q->intlist);
}

(part of the implementation file queue.cpp)

Edit function specifications so they agree with the header

- remove queue_t * q for each method
Step 3: The class:: Syntax

```cpp
class queue {
    public:
        LL_t * intlist;
        int isEmpty();
        void enqueue(int data);
        int dequeue();
    };

queue * queue_create(void);
void queue_destroy(queue * q);
```

(Header file queue.h)

```cpp
int queue_isEmpty() {
    return (q->intlist->head == NULL);
}

void queue_enqueue(int data) {
    LLappend(q->intlist, data);
}

int queue_dequeue() {
    return LLremoveHead(q->intlist);
}
```

(Part of the implementation file queue.cpp)

Edit function specifications so they agree with the header

- remove `queue_t * q` for each method
- instead of the prefix `queue_`, use `queue::`
Step 3: The class:: Syntax

```cpp
class queue {
    public:
        LL_t * intlist;
        int isEmpty();
        void enqueue(int data);
        int dequeue();
    }

queue * queue_create(void);
void queue_destroy(queue * q);
```

(header file queue.h)

```cpp
int queue::isEmpty() {
    return (intlist->head == NULL);
}

void queue::enqueue(int data) {
    LLappend(intlist, data);
}

int queue::dequeue() {
    return LLremoveHead(intlist);
}
```

(part of the implementation file queue.cpp)

Edit function specifications so they agree with the header

- remove `queue_t * q` for each method
- instead of the prefix `queue_`, use `queue::`

Remove the usage of `q->`

- `intlist` is a member of the class and is treated like a local variable
Some Terminology

A *class* encapsulates data with its functions

- data members → *properties*
- function members → *methods*

An *object* is a specific instance of a class

- E.g., The **class of all cars** have defining properties:
  - colour, make, year, mileage, oil level, etc.
  - *but your car* has specific values:
    - white, Matrix, 2008, 145 000 km, 52% oil, etc.

Creation of an object is known as *instantiation.*
To protect class members from external access, use the `private:` keyword

**Rule: All data members should be kept private.**

- Operations on them should be possible only via methods
- This is not merely a matter of style. It’s a matter of:
  - information hiding
  - code independence
The Rhythm of \_create() Functions

Every \_create() function in C so far has followed two steps:

- **allocate** new space on the heap
- **initialize** the data members

In C++, these two steps are separate:

- allocate an object on the heap using the \texttt{new} keyword OR . . .
- declare an object locally
- All created objects are initialized by running a special method called a \texttt{constructor}

```c
queue * queue_create() {
    queue * ret = malloc(sizeof(queue));
    if (ret != NULL) {
        ret->intlist = LLcreate();
    }
    return ret;
}
```

`(queue.c)`

```c
queue * Q = queue_create();
```

`(driver.c)`

```c
queue * Q = new queue; // heap decl.
queue Q; // local declaration
```

`(options for driver.cpp)`
Step 5: Add the Constructor

The constructor method has the same name as the class.

The constructor is always invoked upon instantiation.

- initialize all data members

A constructor can take parameters.
Step 6: Add the Destructor

Destructor is always invoked upon object’s destruction.

- either delete operation OR ...
- local variable goes out of scope

Clean-up procedure for any resources the object held.

```
class queue {
    private:
        LL_t * intlist;
    public:
        queue();
        int isEmpty();
        void enqueue(int data);
        int dequeue();
    };

queue_destroy(queue * q); // (queue.h)
```

```
class queue {
    private:
        LL_t * intlist;
    public:
        queue();
        int isEmpty();
        void enqueue(int data);
        int dequeue();
    };

queue::~queue(); // destructor
LLdestroy(intlist);
}
```

new and delete are the inverses of each other