Last Time

- Talked about Basic C
Today

- Going to talk about C/C++

C++
no need for typedefs for structs

```c
struct Pt {
    int x;
    int y;
};

Pt p = { 1, 2 };```
extern "C" int foo(int x, char* s);
for (int i = 0; i < 10; i++) { ... }
const int CAPACITY = 16;
Basic Concepts

- classes and objects – like C struct but with functions
- member functions – part of object
- private vs public members – access control
Classes

- like C structs
- C++ structs are like a class with only public members
- usually .h file has class defs and .cpp files has implementation
- inside member functions can refer to member fields by name

```cpp
const int CAPACITY = 16;

class Queue {
public:
    void enq (int value);
    int deq ();
    int head;
    int tail;
    int data[CAPACITY];
};

void Queue::enq (int value) {
    data[tail] = value;
    tail = (tail + 1) % CAPACITY;
}
```
use `new` for object creation

use `->` for calling member functions

```cpp
queue = new Queue();
queue->enq(17);
```
- use public for API functions
- use private for hidden fields and functions

```cpp
class Queue {
public:
    void enq (int value);
    int deq ();
private:
    int head, tail;
    int data[CAPACITY];
};
```
Constructors

- use same class name for constructor
- always define a constructor
- always initialize all fields

```cpp
class Queue {
public:
  Queue (int cap);
  void enq (int value);
  int deq ();
private:
  int CAPACITY;
  int head, tail;
  int *data;
};

Queue::Queue(int cap) {
  head = tail = 0;
  CAPACITY = cap;
  data = new int[cap]; // Array construction
}
Destructors

- use ~class name for destructor name
- define a destructor when have allocated data in fields
- also destructors can be used to close files etc
- if don’t write destructor will create leaks

```cpp
class Queue {
public:
    void enq (int value);
    int deq ();
    Queue(int cap);
    ~Queue();
private:
    int CAPACITY;
    int head, tail;
    int *data;
};

Queue::~Queue() {
    delete [] data;
}
```
- declare object using class name
- put class arguments after variable name
- use dot for accessing functions
- destructor called when exiting scope

```cpp
Queue q(10);
q.enq(17);
```
Inline Functions

- can write member function bodies in class definitions
- functions then are inlined

```cpp
class Queue {
public:
    void enq (int value);
    int deq ();
    int capacity () {
        return CAPACITY;
    }
    Queue(int sz);
    ~Queue();
private:
    int CAPACITY;
    int head, tail;
    int *data;
};
```
Call by value

- copies value
- use dot notation

```cpp
class Wrap {
  public:
    int value;
    Wrap(int a) { value = a; }
};

Wrap f(Wrap x, Wrap y) {
  return Wrap(x.value + y.value);
}
```
Inheritance

- inherit interface defined using virtual functions

```cpp
class Queue {
public:
    virtual void enq (int value) = 0;
    virtual int size () = 0;
};
class CircularQueue : public Queue {
public:
    CircularQueue(int cap);
    virtual void enq(int value) {
        data[tail] = value; tail = (tail + 1) % CAPACITY;
    }
    virtual int size() {
        return ((tail - head) + capacity) % CAPACITY;
    }
private:
    int CAPACITY;
    int head, tail;
    int *data;
};
```
Shared Inheritance

- inheritance allows sharing implementation

```cpp
class Queue {
public:
    virtual void enq (int value) = 0;
    virtual int size () = 0;
    bool is_empty () { return size() == 0; } // shared across implementations
};
class CircularQueue : public Queue {
public:
    CircularQueue(int sz);
    virtual void enq(int value) { ... }
    virtual int size() { ... }
private:
    int CAPACITY;
    int head, tail;
    int *data;
};
```
```cpp
template <typename T>
class Queue {
public:
    void enq (T value);
    T deq ();
    int capacity () {
        return CAPACITY;
    }
    Queue(int cap);
    ~Queue();
private:
    int CAPACITY;
    int head, tail;
    T *data;
};
```
template <typename T, int CAPACITY>
class Queue {
public:
    void enq (T value);
    T deq ();
    int capacity () {
        return CAPACITY;
    }
    Queue(int cap);
    ~Queue();
private:
    int head, tail;
    T data[CAPACITY];
};
- defining
- using

```cpp
namespace collections {
    class Queue { ...}
}

queue = new collections::Queue<int>(10);
```

```cpp
using namespace collections;

queue = new Queue<int>(10);
```
can redefine meaning of operators for particular classes

```cpp
class Vec2f {
public:
    float x, y;
    Vec2f operator + (Vec2f o) {
        return Vec2f(x + o.x, y + o.y);
    }
    Vec2f (float ax, float ay) { x = ax; y = ay; }
};

Vec2f u(0.0, 1.0);
Vec2f w(0.0, 1.0);
Vec2f v = u + w;
```
overload normal reading and writing operations on variables
can do for [] as well

class Wrap {
public:
    float x;
    operator int() { return x; }
    operator = (float n) { x = n; }
    Param (float ax) { x = ax; }
};

Wrap p(0.0);
p = 1.0;
float v = p;
callbacks

```c
int f (int a, float b) { ... }
```

```c
typedef int (*int_function_ptr)(int, float);
```

```c
int main () {
    int (*fp)(int, float);
    fp = &f;
    int x = fp(1, 2.0);
}
```

```c
int main () {
    int_function_ptr fp;
    fp = &f;
    int x = fp(1, 2.0);
}
```
String, Set, Map, Vector, Stack, PriorityQueue
mostly discouraged in embedded applications because of memory
dynamic memory allocation – fragmentation and pauses
also blows up code size if that matters
there are some emerging real time options – more later

```cpp
#include <vector>
int main () {
    std::vector v<Int>();
    v.add(1);
    v.add(2);
    int sum = 0;
    for (int i = 0; i < v.size(); i++) {
        sum = sum + v[i];
    }
}
```
C++ compiler

- use same as C compiler

```plaintext
> g++ -o xs xs.cpp  # compiling and linking
```

separate compilation

```plaintext
> g++ -c xs.cpp  # compiling into xs.o
> g++ -o xs xs.o  # linking into xs
```

libraries

```plaintext
> g++ -o xs xs.cpp -lm  # link in math library
```
mBED
A Quick Introduction to C++ by Tom Anderson
Effective C++ by Scott Meyers
Effective Modern C++ by Scott Meyers
STM32L432 Data Sheet
Nucleo32 L432