CS100 Introduction to Programming

Recitation 4

llk89

NO PLAGIARISM!!!

- The most likely cause for failing this course.
- You WILL be caught!
- We WILL punish!
- They WILL know!
  - Parents
  - University
  - School
  - Fellows

Procedure to ask question

- Take the first one that is valid. Do not simply skip to the last.
  1. Ask on piazza, unless the problem
     - Is highly sensitive, e.g. involve your personal medical info
     - Is security related, e.g. you found a security breach in OJ
     - Is of utter urgency, e.g. you cannot attend the exam beginning at next hour
     - Requires examine your code, e.g. a concurrency bug
  2. Ask me (not other TAs), unless the problem
     - Is about homework specification. I have little idea about how it is designed.
     - Is to complain about me. (I’m so kind and helpful why are you doing this)
  3. Ask Wu Tian Yuan or Liu Yu Qi, if the problem is about homework
  4. Ask Wang Jin Rui, if the problem is about recitation designing
  5. Write email to professor
- no response in time => do next action valid

Procedure to ask question

- We can ignore you outright…
  - If you skip some steps in the list.
  - If you ask other recitation TAs (unless being told so)
  - If you ask me a question with an answer on piazza
  - If you ask what test case is
- Not every weird behavior in your code requires examine your code
  - Try to recreate a MVCE
  - Try not to expose too much core logic

QA time

- If you have questions about anything in Lecture 9, ask now
- If you believe you need someone guide you through the lecture step by step, ask now

Overview

- Class constructs
- Constructor & Destructor
- Scopes
- Access modifiers
- Overload
- Object relations
- Hands on practice

Class constructs

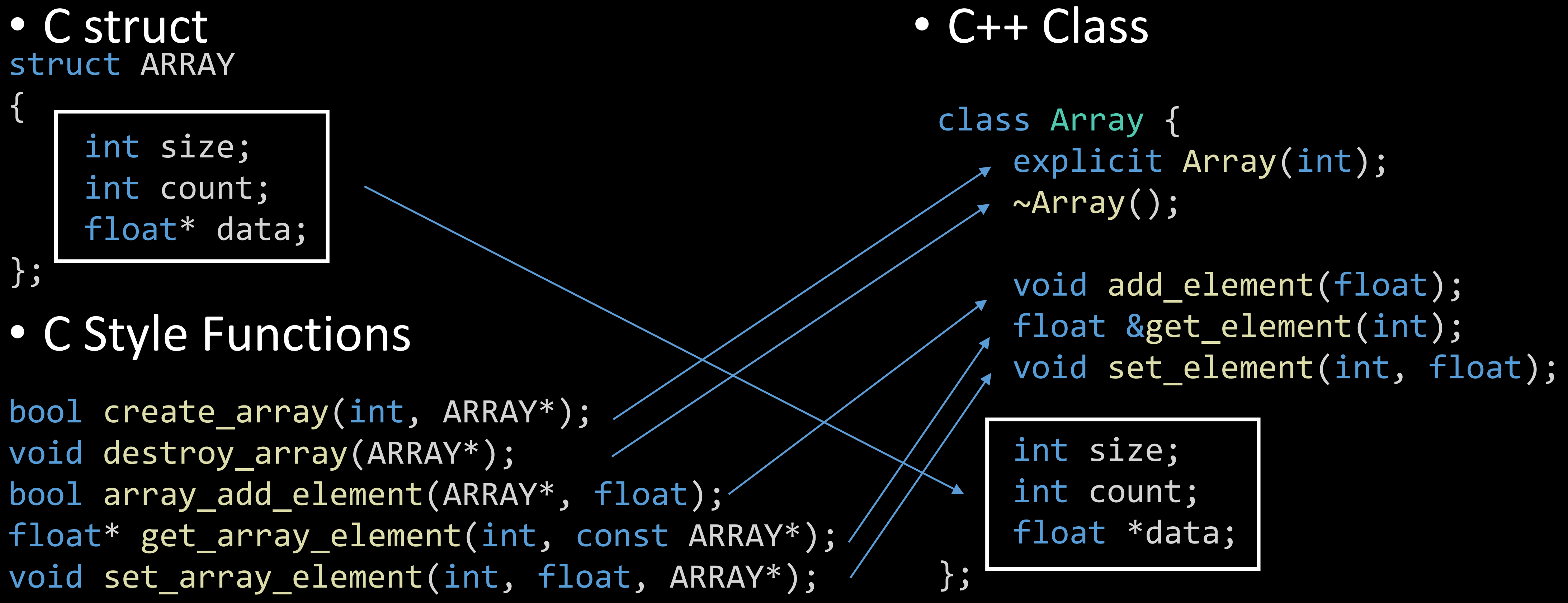
References

- Can be considered a safer pointer
  - Does not need indirection or take address operators
  - Various compiler enforced restrictions, e.g.
    - There can not be a pointer to a reference: int&* is not a valid type
    - Cannot be null
    - Cannot be a dangling pointer unless forced
- Can have reference of references, just like pointers

From struct to class

- Naïve idea: struct with functions
- struct: data
- class: data + behavior
    - Data is defined by fields.
    - Behavior is defined by member functions
    - Special behavior
        - initialization: contructor, allocate resource, set default value, etc
        - destruct: destructor, free allocated resource

Typical transformation



- C struct

```
struct ARRAY
{
    int size;
    int count;
    float* data;
};
```

- C Style Functions

```
bool create_array(int, ARRAY*);
void destroy_array(ARRAY*);
bool array_add_element(ARRAY*, float);
float* get_array_element(int, const ARRAY*);
void set_array_element(int, float, ARRAY*);
```

- C++ Class

```
class Array {
    explicit Array(int);
    ~Array();

    void add_element(float);
    float &get_element(int);
    void set_element(int, float);

    int size;
    int count;
    float *data;
};
```

This transformation isn't complete yet. We will see why later.

this pointer

- Cannot be assigned to
- Point to object itself
  - Therefore not available in static member functions
  - Not null unless serious problem
- Mainly used to distinguish class member and function parameters with the same name
- Some coding styles mandate the use of this pointer before every class member usage to make it clearer that this is a class member

struct in C++

- It has inheritance
- It has member functions
- It has constructors
- Effectively class except default access level is public

Constructor & Destructor

Initializer list

- Initialize members using parameters given

```
class FileStream {
public:
    Foo(const char *file_name) : file(strdup(file_name)) {}
private:
    char *file;
};
```

Call parent constructor

- Similar to initializer list, just replace member name with parent class name.

```
class Foo {
public:
    Foo() {}
};

class Bar : public Foo {
public:
    Bar() : Foo() {}
};
```

Default constructors

- The constructor without any parameter

```
int main() {
    Array i;
    // ...
}
```

- The above code does invoke the default constructor, even if there is no parenthesis

Copy constructors

- Create a copy of current object
- Signature looks like this

```
Array(const Array &);
Array(Array &);
```

  - Notice the parameter is a reference. Why?
  - These two copy constructors looks similar, are they the same?

Destructors

- Release resources associated with current object
- e.g.
    - heap memory
    - files opened
    - database connections
    - ……

Scopes

Scopes

- Learned so far
  - Block scope
  - Function parameter scope (also function scope)
- Also present in C
- C++ has much more
  - Class scope
  - Namespace scope
  - Enumeration scope
  - Template parameter scope
  - Point of declaration
- We will only cover class scope now

Class scope

- Every member declared in a class is visible to everything declared after that member
- Use before declaration is undefined behavior.
- Ambiguity
  - Least favored except items declared outside the class
  - Use this->member to override the rule
  - Use ParentClass::member in subclass to override the rule

Access modifiers

public & private

- Public members are accessible by every function
- Private members are only accessible by every function within the same class scope
- Why?
  - This does NOT improve security against hackers (in most cases)
  - Primarily promote isolation and encapsulation
  - Improve extensibility

public & private

- Public members are expected to stay there
  - It will be a breaking change to remove/modify it
  - Common practice is to notice users three major versions in advance before this change
  - Will be troublesome
- Private members may be removed or modified without notice
  - Code smell if you depends on a private member of a library

Protected

- Certain member may
  - needed by subclasses
    - This means private does not suit here
  - should not be exposed to client code
    - This means public does not suit here
- Solution: use protected
  - Protected member are accessible by subclass and members of this class
  - Protected member are not accessible by any other classes

Overload

Typical usage

- How to write a set of function that print int, float and a number in a string?

```
void print_number(float f) {
    printf("%.3f", f);
}

void print_number(int i) {
    print_number((float) i);
}

void print_number(const char *number) {
    print_number(strtof(number, nullptr));
}
```

Compile Error

NULL vs nullptr

- NULL: typically defined as (void *) 0
  - Definitely a pointer
  - Can be implicitly converted to int
  - Cause overload resolution to fail
- Solution: use nullptr instead
  - Supported since C++11
  - Keyword
  - Drop in replacement of NULL

Code smell

- It may be tempting to write code like this

```
int println(string content, char end);
int println(string content) {
    return println(content, '\n');
}
```

- Use default parameters instead

```
int println(string content, char end = '\n') {
    //...
}
```

Object relations

Inheritance

- When to use inheritance?
  - A concept is an extension to another concept
  - e.g. I/O stream and console stream
  - e.g. Shape and Polygon
  - e.g. Database and SQLite

Composition

- When to use composition
  - A concept can make use of another concept
  - e.g. I/O stream and characters
  - e.g. Polygon and vertices
  - e.g. Database and data type

Inheritance & composition

- Sometimes both applicable

```
class FileStream  {
public:
    void print(int); // print to file
}

class FileAStream : public FileStream {
public:
    void print(int); // print to file A
}

class FileBStream : public FileStream {
public:
    void print(int); // print to file B
}
```

```
class FileStream {
public:
    // parameter is the file to print to
    FileStream(const char *);
    // print to destination file
    void print(int);
}
```

- Prefer composition over inheritance
- Sometimes the gain is not as obvious
- This improves extensibility

Inheritance

- What is the expected output of following program? Why?

```
class Foo {
public:
    Foo()           { printf("1"); }
    void foo()      { printf("2"); bar();}
    void bar()      { printf("3"); }
};

class Bar : public Foo {
public:
    Bar() : Foo() { printf("4"); }
    void foo()    { printf("5"); }
    void bar()    { printf("6"); Foo::bar(); foo(); }
};
```

```
int main() {
    Bar bar; bar.foo(); bar.bar();
}
```

Hands on Practice

Instructions

- Put your hand on keyboard while I walk you through
- Listen to my explanation on why, when and how
- Raise your hand when you have a question

- Write nothing you will learn nothing
- Hear nothing you will learn nothing
- Ask nothing you will learn nothing

Today’s menu

- Implement a C++ object based wrapper around printf and scanf
  - Essentially a naïve version of std::iostream

What now?

- Open your IDE
- Follow my words

Supplementary Info

Preface

- Very few on Earth dares to claim proficiency in C++
- The long history has led to hundreds if not thousands of tiny pieces of strange/awkward/minor feature.
- They may or may not be helpful
- Download slides after recitation to see what are these

Member functions

- Try explain all these const
    - not all are meaningful

```
class Foo {
    const Foo &bar(const int*, const double) const;
};
```

- How about these static?

```
class Foo {
    static FooRegistry registry;
    static Foo &get_foo (const int*, const double);
};
```

Move constructors

- A move constructor of class T is a non-template constructor whose first parameter is T&&, const T&&, volatile T&&, or const volatile T&&, and either there are no other parameters, or the rest of the parameters all have default values.

```
Array(const Array &&);
```

- Typically take the allocated resources from given parameter
- Available since C++11
- See more: https://en.cppreference.com/w/cpp/language/move_constructor

Implicit implementation

- Trivial copy constructors, move constructors and destructors look highly similar
- Just numerous copy/move/delete
- Tedious to write them all
- Compiler will generate one if there is no user-defined one.
- Force compiler to generate one even if user defined another

```
Array(const Array &) = default;
```

- Force compiler not to generate one even if no user-defined one found

```
Array(const Array &) = delete;
```

Implicit conversion

- This declaration looks innocent

```
class Array {
public:
    Array(size_t size);
}
```

- Until you realize this is valid C++

```
Array a = 3;
```

- Not always desirable
- Use explicit to avoid

```
class Array {
public:
    explicit Array(size_t size);
}
```

friend

- Friend access allow a specific function or class to access specific member, or every member of a class
- Refrain from using unless necessary

```
class Foo {
public:
    Foo(int i) : i(i) {}
private:
    int i;
    friend void print(Foo &);
};

void print(Foo &foo) {
    printf("%d\n", foo.i);
}
```