# CIS 014 – C++ Programming

Lecturer: Joseph Su

PASADENA CITY COLLEGE ®

# REFERENCES

**<u>Optional Textbook:</u>**

*Programming: Principles and Practice Using C++, 2nd ed, B. Stroustrup, Addison-Wesley, 2014*

**PDF:**

http://www.cplusplus.com/files/tutorial.pdf

**Online:**

http://www.cplusplus.com/doc/tutorial/

***The C++ Programming Language, 4th ed.***

*B. Stroustrup, Addison-Wesley, 2013*

**C++ How to Program, 10th ed**

*Deitel & Deitel*, Pearson Hall, 2016

**C++ Primer, 5th ed**

*S. Lippman, J. Lajoie, and B. Moo*, Addison-Wesley, 2012

# READING ASSIGNMENTS

**ONLINE**

- Classes and Structs (C++)
- Dynamic Memory

**REFERENCES**

http://www.cplusplus.com/files/tutorial.pdf (pages 1-99)
http://www.cplusplus.com/doc/tutorial/

- ✓ Program Structure
  - Complete all chapters
- ✓ Compound Data Types
- ✓ Classes
  - Classes I, Classes II

# TODAY

- Review: Constructor & Destructor
- Struct
  - Definition
  - Declaration
  - Pointers
  - Arrays
  - Examples
- Class vs. Struct
- Class:
  - Pointers: Examples
  - Accessors: Examples
  - The this Pointer
- Dynamic Deallocation

# CLASS: CONSTRUCTOR

- Using the `CRectangle` class example.

- At run time you can create an object of class type `CRectangle` via either of the following statements:

```
// Stack allocation
CRectangle rect;

// Dynamic allocation on heap
CRectangle* rect = new CRectangle();
```

- The 2nd statement above invokes `CRectangle`'s default constructor:

```
CRectangle();
```

# CLASS: CONSTRUCTOR EXAMPLE

```
class CRectangle {
    int width, height;
  Public:
    CRectangle();              // default constructor
    CRectangle(int,int);       // user-defined constructor
    ~CRectangle();             // default destructor
    int area() {return (width*height);}
};

// overwrite the default constructor with specific user-
// defined implementation
CRectangle::CRectangle() {
    width = 5;
    height = 5;
}
// user-defined constructor implementation
CRectangle::CRectangle(int a, int b) {
    width = a;
    height = b;
}
```

# CLASS: DESTRUCTOR

- `CRectangle()` default constructor is provided by the compiler if you don't create your own constructor.
- `CRectangle` also has a destructor, which performs the opposite of what a constructor does.
- By default that destructor is

```
~CRectangle();
```

- Note the ~ sign on a destructor, which has the same name as the class name.
- Cannot be declared `static` or `const`.
- Destructor is where all of your dynamic variables are de-allocated

# CLASS: DESTRUCTOR EXAMPLE

- Should be declared in public section of class.
- No return type.
- Automatically invoked when object goes out of scope:
  - Function ends
  - Program ends
  - Delete is called on object
  - Block containing local vars ends

```
class CRectangle {

      public:
         CRectangle();       // default constructor
        ~ CRectangle();     // default destructor
};

CRectangle::~ CRectangle() {
      // dynamically deallocate vars if needed
}
```

# STRUCT: WHAT IS IT?

- Classes can also be defined with the keyword, `struct`
- A way to group member data elements of similar or different types together

```
struct structure_name {
   member_type1 member_name1;
   member_type2 member_name2;
   …
} object_names;
```

**Example:**

```
struct product {
   int weight;
   float price;
} apple, banana, melon;
```

# STRUCT: WHAT IS IT?

- Used by C, a procedural language before the concept of class arrived
- Very similar to class, grouping both member variables and functions together
- But missing a lot of Object-Oriented Programming traits such as inheritance and polymorphism
- Also missing certain functional attributes such as private and protected access in data encapsulation
- Lastly, you cannot declare C++ functions inside a `struct`. Instead you have to use function pointers:

```
struct sample {
   int (*test)(char*);
};
int test(char* c) {…}
```

# STRUCT: EXAMPLE

**Example:**

```
struct product {
    int weight;
    float price;
} apple, banana, melon;
```

- `product` is the structure type
- `apple`, `banana`, and `melon` are instances of this type

- At run time members of the `struct` can be accessed via the . (dot) operator:

```
apple.weight
banana.price
melon.price
…
```

# STRUCT: ARRAY

- Can also create an array of elements of type `struct`

**Example:**

```
const int NUM_FRUITS = 3;

struct product {
    int weight;
    float price;
} fruits [NUM_FRUITS];
```

- At run time members of the `struct` can be accessed via the (dot) operator, '.', and array index:

```
fruits[0].weight
fruits[1].price
```

# STRUCT: POINTERS

- Structure can have pointer pointing to it

**Example:**

```
const int NUM_FRUITS = 3;

struct product {
   int weight;
   float price;
};

product apple;
product* pFruit = &apple;
```

- At run time members of the `struct` can be accessed via the arrow operator (->):

```
pFruit->weight
pFruit->price
```

# CLASS VS. STRUCT

- In C++, a struct is actually a class
  - Difference being that default member and base class access specifiers for struct is PUBLIC, for class is PRIVATE

- In C, a struct cannot have static member variables and static member functions
  - C struct encapsulates data but does NOT implement behaviors
  - C struct can’t provide OOP (inheritance, polymorphism)

# CLASS: POINTERS

```
class Fruit {…};

class FruitBasket {
        Fruit mFruits[10];
    public:
        FruitBasket();
        ~FruitBasket();
        void showFruits();
        void addFruit(Fruit*);
        void removeFruit(Fruit*);
};

int main() {

    FruitBasket* pBasket;

    pBasket = new FruitBasket();

    …

    return 0;
}
```

# CLASS: POINTERS

- Declaring a pointer to the class, `FruitBasket`:

```
FruitBasket* pBasket;
```

- Dynamically allocating memory for `FruitBasket` on the heap:

```
new FruitBasket();
```

The above creates an instance of `FruitBasket`, calls its constructor, and returns a reference to the memory

- Assigning the memory reference to `pBasket`:

```
pBasket = new FruitBasket();
```

# CLASS: POINTERS

- Accessing `FruitBasket`'s constructor:

```
FruitBasket* pBasket = new FruitBasket();
```

- Accessing `FruitBasket`'s `FruitBasket(int, int)` constructor:

```
FruitBasket* pBasket = new FruitBasket(1, 2);
```

- Accessing `FruitBasket`'s public method, `void showFruits()`:

```
pBasket -> showFruits();
```

- Accessing `FruitBasket`'s private variable, `mFruits`:

```
pBasket -> mFruits;        // ERROR!
```

# CLASS: POINTERS

- To access FruitBasket's private variable mFruits, we can create a public accessor method called getFruits():

```
Class FruitBasket {
        …
    public:
        Fruit* getFruits();
        …
};

Fruit* FruitBasket::getFruits() {
    return mFruits;
}
```

Then we may:

```
pBasket -> getFruits();
```

# CLASS: ACCESS EXAMPLES

- Assigning a reference to the `FruitBasket` instance in memory to another pointer:

```
FruitBasket a;
FruitBasket** b;

FruitBasket* pBasket = new FruitBasket();

b = &pBasket;

// accessing the public method in the pBasket instance
(*b) -> getFruits();

// same as above
pBasket -> getFruits();

// accessing the public method in the a instance
a.getFruits();
```

# CLASS: ACCESS EXAMPLES

- Dynamically allocating an array of 10 FruitBasket* instances:

```
FruitBasket** pBaskets = new FruitBasket*[10];

// accessing the pBaskets[0]'s getFruits() method
pBaskets[0] -> getFruits();

// accessing the pBaskets[1]'s getFruits() method
pBaskets[1] -> getFruits();
```

- Dynamically de-allocating the `pBaskets` array:

```
// de-allocating the pBaskets
delete [] pBaskets;
pBaskets = NULL;
```

# CLASS: `this` OPERATOR

- `this` refers to the instantiated object itself
- `this` is a pointer
- `this` is an implicit pointer to all member functions and variables inside the class it belongs to:

```
void FruitBasket::getFruits() {
   //which also works without this ->
   this->mFruits;
}

Versus:

void FruitBasket::getFruits() {
   mFruits;
}
```

# DYNAMIC DEALLOCATION

- Track your pointer to dynamically allocated space

```
// somewhere in your code block
char* p = new char(4);
```

- You may pass `p` around, from functions to functions

```
char* func(char* p) {
    ...
}
```

- You may deallocate that allocated space anywhere, perhaps in another function:

```
char* another_func(char* p) {
    delete p;
    p = nullptr;
}
```

# DYNAMIC DEALLOCATION

- Dynamically allocating an array of 4 integers:

// this creates spaces for each element in the array, namely p[0],
// p[1], p[2], p[3], and p[4] – in array's lifetime order:

```
int* p = new int[5];
```

- Dynamically de-allocating p with approach (1) or approach (2) below:

(1)

// this **only** deletes p[0], leaving p[1], p[2], p[3], p[4] as-is

```
delete p;
```

(2)

// this deletes p[4], p[3], p[2], p[1], p[0] in array's lifetime order

```
delete [] p;
```

# DYNAMIC DEALLOCATION

If you have to use dynamic memory allocation in your code:

- It is IMPORTANT that you track the lifetime of any pointer, beginning with where it points to (dynamically or statically allocated memory block), where it gets used, and to where it gets de-allocated eventually.

- If not deallocated, deallocate it!

- Any dangling pointers pointing to any previously allocated space left at the execution end of your program will result in MEMORY LEAKS!