OOP in C++ - Part 2 and the C Preprocessor

MATH 5061: Fundamentals of Computer Programming for Scientists and Engineers

Dr. Richard Berger
richard.berger@temple.edu

Department of Mathematics
Temple University

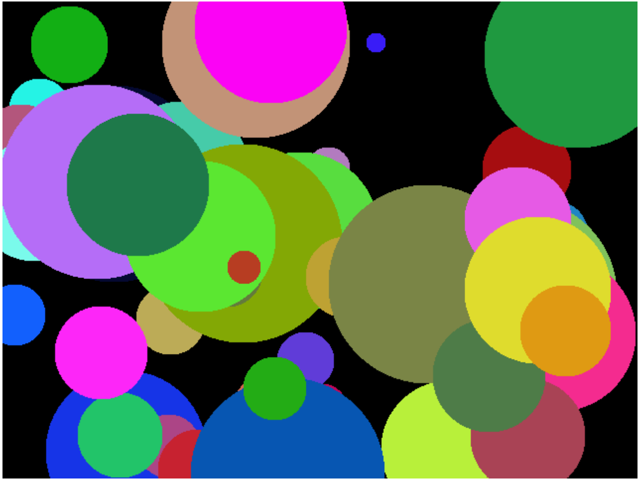
11/08/2016

Outline

TEMPLE UNIVERSITY

Live Demo: Drawing shapes continued

Shapes program output



Code is part of the homework 9 assignment

Outline

- Object-Oriented Programming in C++ - Part 2
  - More Constructor Examples
  - Global variables
  - Static class members
  - Constant class members
  - Copy constructor
  - Call-by-Value
  - Shallow copy vs Deep copy
  - Operator overloading
- Working with multiple files
  - Multi-definition problem
  - C Preprocessor

Constructor/Destructor examples

```
struct Point {
    double x;
    double y;

    Point() : x(0.0), y(0.0) {
        printf("+ Point created at (%f, %f)!\n", x, y);
    }

    Point(double x, double y) : x(x), y(y) {
        printf("+ Point created at (%f, %f)!\n", x, y);
    }

    ~Point() {
        printf("- Point destroyed!\n");
    }
};
```

Constructing inidividual objects

```
int main() {
    Point p1;
    Point p2(1.0, 2.0);
    return 0;
}
```

Output

```
+ Point created at (0.000000, 0.000000)!
+ Point created at (1.000000, 2.000000)!
- Point destroyed!
- Point destroyed!
```

Constructing arrays of objects

Stack objects

```
int main() {
    Point points[10];
    return 0;
}
```

Heap objects

```
int main() {
    Point* points = new Point[10];
    delete [] points;
    return 0;
}
```

Output

```
+ Point created at (0.000000, 0.000000)!
+ Point created at (0.000000, 0.000000)!
+ Point created at (0.000000, 0.000000)!
+ Point created at (0.000000, 0.000000)!
+ Point created at (0.000000, 0.000000)!
+ Point created at (0.000000, 0.000000)!
+ Point created at (0.000000, 0.000000)!
+ Point created at (0.000000, 0.000000)!
+ Point created at (0.000000, 0.000000)!
+ Point created at (0.000000, 0.000000)!
- Point destroyed!
- Point destroyed!
- Point destroyed!
- Point destroyed!
- Point destroyed!
- Point destroyed!
- Point destroyed!
- Point destroyed!
- Point destroyed!
- Point destroyed!
```

Global variables

```c
#include <stdio.h>
#include "point.h"

// global variables
int g = 42;
Point p;

int main() {
    printf("=== beginning of main() ===\n");

    printf("Point p at (%f, %f)\n", p.x, p.y);

    printf("Global g=%d\n", g);

    printf("=== end of main() ===\n");
    return 0;
}
```

- variables which are defined outside of the scope of a function
- visible to all functions or class methods
- they are neither on the stack nor the heap

Warning

Use global variables with extreme caution. Writing code which manipulates a global state is much more error prone. Always prefer local variables and class members.

TEMPLE UNIVERSITY

External global variables

Listing 1: A.cpp

```
#include <stdio.h>
#include "point.h"

// external global variable
extern int g;
extern Point p;

int main() {
    printf("=== beginning of main() ===\n");

    printf("Point p at (%f, %f)\n", p.x, p.y);

    printf("Global g=%d\n", g);

    printf("=== end of main() ===\n");
    return 0;
}
```

Listing 2: B.cpp

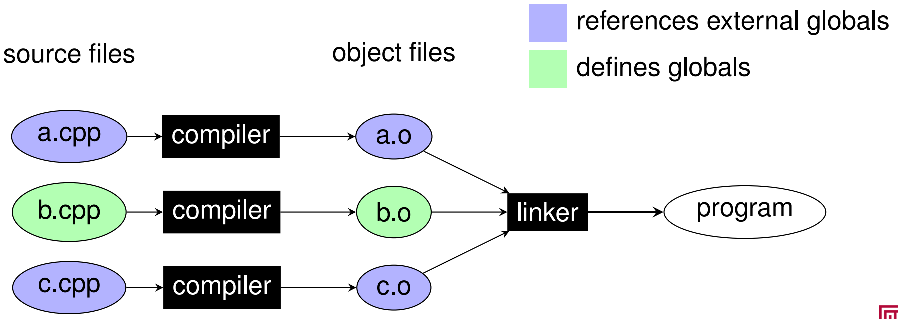
```
#include "point.h"

// global variables
int g = 42;
Point p;
```

- global variables which are defined in some other file must be declared as **`extern`**
- such variables must be defined once in a source file

Recall: Linking

- Each source file is compiled separately by a different compiler instance
- They do not share information
- So a compiler has to add placeholders for external global variables
- These placeholders are then resolved by the linker



Global variable initialization

```
+ Point created at (0.000000, 0.000000)!
=== beginning of main() ===
Point p at (0.000000, 0.000000)
Global g=42
=== end of main() ===
- Point destroyed!
```

- global variables are initialized before the `main` function is called
- they are also destroyed after the main function

Hint

If your program crashes after returning from `main()`, you might have a problem in one of your destructors!

Static member variables

```
struct Point {
    static int nobjects;
    int id;
    double x;
    double y;

    Point() : x(0.0), y(0.0) {
        id = ++nobjects;
    }

    ~Point() {
        --nobjects;
    }
};

// initialize static outside of class
int Point::nobjects = 0;
```

- member variables of classes are stored on a per-object basis
- **static member variables** are stored only **once** per class
- outside of class method code, static member variables are accessed as `ClassName::variable_name`
- they are accessible by all objects of that class and outside if it has **`public`** visibility
- initialization must be outside of the class, just like a global variable

Static member functions

```
struct Point {
    static int nobjects;
    ...

    static void print_number_of_objects() {
        printf("# Points: %d\n", nobjects);
    }
};
```

```
// initialize static outside of class
int Point::nobjects = 0;

int main() {
    Point p1;
    Point::print_number_of_objects();
    return 0;
}
```

- member functions in a class only act on individual objects
- **static member functions** are not associated with an object, but to the class
- they can only access static member variables

TEMPLE UNIVERSITY

Constant variables

```
class Temperature {
    float kelvin;
public:
    Temperature();
    Temperature(float kelvin);

    float get_kelvin();
    float get_celsius();
    float get_fahrenheit();

    float set_kelvin(float value);
    float set_celsius(float value);
    float set_fahrenheit(float value);

    void print();
};
```

```
// constant variable
const int c = 10;
c = 20; // COMPILE ERROR

// constant object
const Temperator t(20);
t.set_kelvin(40); // COMPILE ERROR
```

- the `const` modifier tells the compiler that a given variable should not change its value after its initial definition
- a `const` object should not change state, either by accessing it member variables or calling its member functions

Constant member methods

```
class Temperature {
    float kelvin;
public:
    Temperature();
    Temperature(float kelvin);

    float get_kelvin() const ;
    float get_celsius() const ;
    float get_fahrenheit() const ;

    float set_kelvin(float value);
    float set_celsius(float value);
    float set_fahrenheit(float value);

    void print() const ;
};
```

- many methods do not change the state, but instead only read values
- if that is the case, you can mark those methods as **const**
- a **const** method can be called for a **const** or non-**const** object

```
const Temperator t(20);
float k = t.get_kelvin();     // ok
float c = t.get_celsius();    // ok
float f = t.get_fahrenheit(); // ok
t.print(); // ok
t.set_kelvin(40); // COMPILE ERROR
```

Copy construction

```
int main() {
    Temperature t1;
    t1.set_fahrenheit(0);
    t1.print();

    Temperature t2 = t1;
    t2.print();

    Temperature t3(t1);
    t3.print();

    printf("&t1 = %p\n", &t1)
    printf("&t2 = %p\n", &t2)
    printf("&t3 = %p\n", &t3)
    return 0;
}
```

Copy construction

```
int main() {
    Temperature t1;
    t1.set_fahrenheit(0);
    t1.print();

    Temperature t2 = t1;
    t2.print();

    Temperature t3(t1);
    t3.print();

    printf("&t1 = %p\n", &t1)
    printf("&t2 = %p\n", &t2)
    printf("&t3 = %p\n", &t3)
    return 0;
}
```

```
+ created object 1/1
C: -17.78, F: 0.00, K: 255.37
C: -17.78, F: 0.00, K: 255.37
C: -17.78, F: 0.00, K: 255.37
&t1 = 0x7fffbe1f22a0
&t2 = 0x7fffbe1f2290
&t3 = 0x7fffbe1f2280
- deleted object 1/1
- deleted object 1/0
- deleted object 1/-1
```

- we only see output from one constructor
- but there are 3 objects at different addresses
- and the destructor is called 3 times
- **someone is creating copies for us**

Copy Constructor

```
// custom copy constructor
Temperature(const Temperature & t) {
    kelvin = t.kelvin;
    id = ++nobjects;
    printf("+ created copy %d/%d\n",
           id, nobjects);
}
```

- by default, the compiler will create a **copy constructor**
- this constructor makes a binary copy of the original object when a new object is created using either assignment or construction with an existing object
- you can define your own copy constructor with the name `ClassName(const ClassName &)`

Copy Constructor

```
// custom copy constructor
Temperature(const Temperature & t) {
    kelvin = t.kelvin;
    id = ++nobjects;
    printf("+ created copy %d/%d\n",
           id, nobjects);
}
```

```
+ created object 1/1
C: -17.78, F: 0.00, K: 255.37
+ created copy 2/2
C: -17.78, F: 0.00, K: 255.37
+ created copy 3/3
C: -17.78, F: 0.00, K: 255.37
&t1 = 0x7ffc3282bec0
&t2 = 0x7ffc3282beb0
&t3 = 0x7ffc3282bea0
- Deleted object 3/3
- Deleted object 2/2
- Deleted object 1/1
```

```
class Celsius : public Temperature {
public:
    Celsius(float celsius) {
        set_celsius(celsius);
        printf("++ Celsius constructor: %d/%d\n", id, nobjects);
    }
};
```

- Q: How many objects are created in the following code and how long do they live?

```
int main() {
    Temperature t1;
    t1.set_fahrenheit(0);
    t1.print();

    Temperature t2 = Celsius(60);
    t2.print();
    return 0;
}
```

Temporary objects

```
+ Temperature: Default constructor created object 1/1
C: -17.78, F: 0.00, K: 255.37
+ Temperature: Default constructor created object 2/2
++ Celsius constructor: 2/2
+ Temperature: Copy Constructor created object 3/3
- Deleted object 2/3
C: 60.00, F: 140.00, K: 333.15
- Deleted object 3/2
- Deleted object 1/1
```

- A: Although only 2 temperature variables are declared, 3 objects are created

1. the `Celsius` initialization first calls the base constructor of `Temperature`
2. this increases the `nobject` counter
3. then the `Celsius` constructor is executed. This is still the same object #2
4. this fully initialized **temporary** `Celsius` object is then assigned to the `Temperature` variable
5. that assignment will cause a **copy construction** of a new `Temperature` object to occur
6. finally the `Celsius` object ceases to exist immediately

```
Temperature(const Temperature & t);
```

- Q: Why can a `Celsius` object be used for copy construction of a `Temperature` object?

```
Temperature(const Temperature & t);
```

- Q: Why can a `Celsius` object be used for copy construction of a `Temperature` object?
- A: Because `Celsius` is a subclass of `Temperature` and therefore the reference type of this base class is compatible.

```
Celsius c(20); // Celsius is a subclass of Temperature

Temperature & ref = c; // this works

// so this is possible
Temperature t(c);
```

Call-by-Value

- So far we've passed larger objects to functions via pointers or references

```
void print_kelvin_sum(Temperature & a, Temperature & b) {
    float sum = a.get_kelvin() + b.get_kelvin();
    printf("&a = %p\n", &a); // print address of a
    printf("&b = %p\n", &b); // print address of b
    printf("sum: %2.2f\n", sum);
}
```

```
Temperature t1(20);
Temperature t2(40);
print_kelvin_sum(t1, t2);
```

TEMPLE UNIVERSITY

Call-by-Value

- So far we've passed larger objects to functions via pointers or references

```
void print_kelvin_sum( Temperature & a , Temperature & b ) {
    float sum = a.get_kelvin() + b.get_kelvin();
    printf("&a = %p\n", &a); // print address of a
    printf("&b = %p\n", &b); // print address of b
    printf("sum: %2.2f\n", sum);
}
```

```
Temperature t1(20);
Temperature t2(40);
print_kelvin_sum(t1, t2);
```

- Q: What happens if we do not use pointer or reference types?

```
void print_kelvin_sum( Temperature a , Temperature b ) {
    float sum = a.get_kelvin() + b.get_kelvin();
    printf("&a = %p\n", &a); // print address of a
    printf("&b = %p\n", &b); // print address of b
    printf("sum: %2.2f\n", sum);
}
```

Output

```
+ Temperature: Parameter constructor created object 1/1
+ Temperature: Parameter constructor created object 2/2
&t1 = 0x7fffbfa38a00
&t2 = 0x7fffbfa389f0
+ Temperature: Copy Constructor created object 3/3
+ Temperature: Copy Constructor created object 4/4
&a = 0x7fffbfa38a20
&b = 0x7fffbfa38a10
sum in kelvin: 60.00
- Deleted object 4/4
- Deleted object 3/3
- Deleted object 2/2
- Deleted object 1/1
```

Copy-by-value

- just like primitive data types (**`int`**, **`float`**, etc) passing objects directly as function parameters creates a copy on the new stack frame of the called function
- this copy operation is done by the copy constructor
- the default copy constructor creates a binary copy of the data
- if you do not want this behavior you have to define your own copy constructor

```
void print_sum(Temperature a, Temperature b)
{
    float sum=a.get_kelvin()+b.get_kelvin();
    printf("&a = %p\n", &a);
    printf("&b = %p\n", &b);
    printf("sum: %2.2f\n", sum);
}

int main() {
    Temperature t1(20);
    Temperature t2(40);

    printf("&t1 = %p\n", &t1);
    printf("&t2 = %p\n", &t2);

    print_sum(t1, t2);

    return 0;
}
```

Stack Frame: `main()`

t1 = Temperature #1 (20 K)
t2 = Temperature #2 (40 K)

```
void print_sum(Temperature a, Temperature b)
{
    float sum=a.get_kelvin()+b.get_kelvin();
    printf("&a = %p\n", &a);
    printf("&b = %p\n", &b);
    printf("sum: %2.2f\n", sum);
}

int main() {
    Temperature t1(20);
    Temperature t2(40);

    printf("&t1 = %p\n", &t1);
    printf("&t2 = %p\n", &t2);

    print_sum(t1, t2);

    return 0;
}
```

Stack Frame: `main()`

t1 = Temperature #1 (20 K)

t2 = Temperature #2 (40 K)

Stack Frame: `print_sum()`

b = Temperature #4 (40 K)

a = Temperature #3 (20 K)

```c
void print_sum(Temperature a, Temperature b)
{
    float sum=a.get_kelvin()+b.get_kelvin();
    printf("&a = %p\n", &a);
    printf("&b = %p\n", &b);
    printf("sum: %2.2f\n", sum);
}

int main() {
    Temperature t1(20);
    Temperature t2(40);

    printf("&t1 = %p\n", &t1);
    printf("&t2 = %p\n", &t2);

    print_sum(t1, t2);

    return 0;
}
```

Stack Frame: `main()`

t1 = Temperature #1 (20 K)

t2 = Temperature #2 (40 K)

Stack Frame: `print_sum()`

b = Temperature #4 (40 K)

a = Temperature #3 (20 K)

sum = 60.0

```
void print_sum(Temperature a, Temperature b)
{
    float sum=a.get_kelvin()+b.get_kelvin();
    printf("&a = %p\n", &a);
    printf("&b = %p\n", &b);
    printf("sum: %2.2f\n", sum);
}

int main() {
    Temperature t1(20);
    Temperature t2(40);

    printf("&t1 = %p\n", &t1);
    printf("&t2 = %p\n", &t2);

    print_sum(t1, t2);

    return 0;
}
```

Stack Frame: `main()`

t1 = Temperature #1 (20 K)
t2 = Temperature #2 (40 K)

Stack Frame: `print_sum()`

```
void print_sum(Temperature a, Temperature b)
{
    float sum=a.get_kelvin()+b.get_kelvin();
    printf("&a = %p\n", &a);
    printf("&b = %p\n", &b);
    printf("sum: %2.2f\n", sum);
}

int main() {
    Temperature t1(20);
    Temperature t2(40);

    printf("&t1 = %p\n", &t1);
    printf("&t2 = %p\n", &t2);

    print_sum(t1, t2);

    return 0;
}
```

Stack Frame: `main()`

t1 = Temperature #1 (20 K)
t2 = Temperature #2 (40 K)

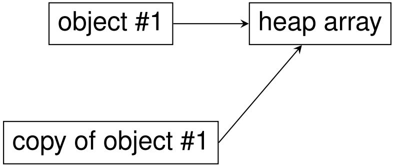
Shallow copy vs. Deep Copy

Shallow copy

- Copy-by-value introduces a problem if your objects contain pointers
- Binary copies of objects lead to copies of all pointers inside
- Different objects, pointing to the same data

New Questions:

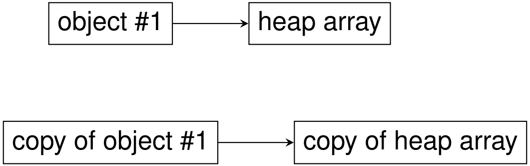
- Which object owns the data on the heap?
- Who is responsible for clean up during destruction?



Shallow copy vs. Deep Copy

Deep copy

- By defining a custom copy constructor you can take care of this situation by duplicating all internal data
- This duplication creates a so-called deep copy



Operator Overloading

```
int main() {
    Temperature t1(20);
    Temperature t2(40);

    Temperature t3 = t1 + t2;
    t3.print();

    return 0;
}
```

- operator overloading allows to define what happens if operators act on objects of a class
- in this example we want to create a new `Temperature` object which stores the sum of two other temperatures
- operators can be either defined using member functions or as global functions

Overloading an operator with a member function

```
class Temperature {
    ...
    // operator as member function
    Temperature operator+(const Temperature & right) {
        return Temperature(kelvin + right.kelvin);
    }
};
```

- one benefit of using a member function is that we have direct access to private member variables, such as `kelvin`

Overloading an operator using a global function

```
// operator as non-member function
Temperature operator+(const Temperature & a,
                      const Temperature & b)
{
    return Temperature(a.get_kelvin() + b.get_kelvin());
}
```

- regular functions are still very useful, however we do not have direct access to members anymore

Outline

Multi-definition problem

- Splitting up code into multiple files is useful to distribute complexity and make code more readable
- headers are used to include definitions of global variables and data types
- however sometime it can happen that you include a header more than once due to dependencies (e.g. multiple subclasses in different files)

Multi-definition problem: Example

base.h

```
class Base {
};
```

derived_a.h

```
#include "base.h"

class DerivedA : public Base {
};
```

derived_b.h

```
#include "base.h"

class DerivedB : public Base {
};
```

main.cpp

```
#include "derived_A.h"
#include "derived_B.h"

int main() {
    ...
}
```

Problem

- base.h is included twice in main.cpp
- class `Base` is then defined twice in main.cpp

C Preprocessor

- C++ uses the C Preprocessor during before each compilation
- simple text manipulation tool
- controlled by directives starting with #
- **`#include`**-directive is one of them

Looking at preprocessor output

You can look at the output of the preprocessor stage of GCC by using the -E flag

```
g++ -E main.cpp > main_preprocessed.cpp
```

Simple text replacement

```
#define NAME VALUE
```

```
a = NAME;
```

becomes

```
a = VALUE;
```

- in older C/C++ codes often used to define constants

  ```
  #define PI 3.14
  ```

Optional code parts

Add one text or another based on defined values

```
#ifdef NAME
// if NAME is defined with any value, use this text
#else
// otherwise use this text
#endif
```

```
#ifndef OTHER_NAME
// if OTHER_NAME is NOT defined with any value, this text
#else
// otherwise use this text
#endif
```

Include Guards

```
#ifndef SOME_HEADER_H__
#define SOME_HEADER_H__

// add definitions only once

#endif
```

- include guards ensure that definitions inside are only included once
- this avoid multiple definitions

Multi-definition problem: Resolution with C Preprocessor

base.h

```
#ifndef BASE_H__
#define BASE_H__

class Base {
    ...
};

#endif
```

Multi-definition problem: Resolution with C Preprocessor

derived_a.h

```
#include "base.h"

#ifndef DERIVED_A_H__
#define DERIVED_A_H__

class DerivedA : public Base {
    ...
};

#endif
```

derived_b.h

```
#include "base.h"

#ifndef DERIVED_B_H__
#define DERIVED_B_H__

class DerivedB : public Base {
    ...
};

#endif
```