CS100
Introduction to Programming

Lecture 9. Classes, Function Overloading and Inheritance

An Overview of Class

- **An object-oriented augmentation of structure**
  - Member variables
  - Member functions
  - Access specifiers
  - Constructor/destructor

```
struct person
{
    char* name;
    int age;
    float height;
    float weight;
};
```

```
class person
{
public:
    void set_name(char* name);
    char* get_name();
    void set_age(int age);
    int get_age();
    void set_height(float height);
    float get_height();
    void set_weight(float weight);
    void get_weight();

    person();
    person(const char* name, int age,
                float height, float weight);
    person(const Person& p);
private:
    char* name;
    int age;
    float height;
    float weight;
};
```

Class Object Initialization

- **When creating a class object**
  - A constructor is called
  - Which constructor to call depends on parameters

```
person p1;
person p2("Li Min", 20, 175, 65);
person p3(p2);
```

static construction

```
person* p_p1 = new person();
person* p_p2 = new person("Li Min", 20, 175, 65);
person* p_p3 = new person(*p_p2);
...
delete p_p1;
delete p_p2;
delete p_p3;
```

dynamic construction

Class Object Initialization

- **When creating a class object**
  - What do “new” and “delete” do?

```
person* p_p2 = new person("Li Min", 20, 175, 65);
```

```
person* p_p2 = (person*)malloc(sizeof(person));
p_p2->person("Li Min", 20, 175, 65); //illegal, for illustration only
```

```
delete p_p2;
```

```
p_p2->~person(); //illegal, for illustration only
free(p_p2);
```

Constructor with Default Parameters

- Look again the two constructors

```
class person
{
public:
     ...

     person();
     person(const char* name, int age,
                 float height, float weight);
     person(const Person& p);
private:
     char* name;
     int age;
     float height;
     float weight;
};
```

Constructor with Default Parameters

- **They can be combined into one constructor**
  - With default parameters

```
class person
{
public:
    ...

    person(const char* name = NULL, int age = 0,
           float height = 0.0f, float weight = 0.0f);
    person(const Person& p);
private:
    char* name;
    int age;
    float height;
    float weight;
};
```

```
person p1;
person p2("Li Min", 20, 175, 65);
```

More on Copy Constructor

- **Copy the entire content of another object**
  - The object is of the same class type
  - Can have free access to the member of the object being copied

```
class line
{
public:
     char* get_string();

     line(const char* str);
     line(const line& str);
     ~line();
private:
     char* line_str;
     int size;
};
```

```
line::line(const line& str)
{
     if (strlen(line_str)
                    != strlen(str.line_str)){
          delete []line_str;
          line_str =
             new char[strlen(str.line_str) + 1];
     }

     if (line_str != NULL){
          strcpy(line_str, str.line_str);
          size = str.size;
     }
     else
          size = 0;
}
```

Class Object Destruction

- **Destructor of a class object**
  - Called when object is deleted from memory
    - Usually used to clean up dynamically allocated memories within the object

```
class mem_alloc
{
public:
    float* get_data();
    float* create(int count);
    float* destroy();

    mem_alloc(int count);
    ~mem_alloc();
private:
        float* m_data;
};
```

```
mem_alloc::mem_alloc(int count)
{
    m_data = new float[count];
}

mem_alloc::~mem_alloc()
{
    if (m_data != NULL)
        delete []m_data;
}
```

Default Constructor

- **What if you do not write any constructor(s) or destructor?**
  - The compiler will generate them for you
  - They do nothing but an empty function

```
class vertex
{
public:
    float x, y, z;

    vertex& operator = (const vertex&);
    float get_dist();//get distance to origin
};
```

No constructor and destructor declared

```
vertex::vertex()
{

}

vertex::~vertex()
{

}
```

More on Access Specifiers

- **We have three different options for *access specifiers*, each with their own role:**
  - public: fully accessible by an object
  - private: accessed within the object
  - protected: very restricted access by an object
- **Access with class object**
  - Only public members (data & functions)

```
person p("Li Min",
        20, 175, 65);
printf("name: %s\n",
        p.get_name());
```

```
person *p_p("Li Min",
        20, 175, 65);
printf("name: %s\n",
        p_p->get_name());
```

Classification of Member Functions

- Many classifications
- Typical classification
  - Accessor functions
  - Mutator functions
  - Auxiliary functions

```
class person
{
public:
     void set_name(char* name);
     void set_age(int age);
     ...
     char* get_name();
     int get_age();
     ...
     void print();

     person();
     person(const char* name, int age,
                  float height, float weight);
     person(const Person& p);
private:
     char* name;
     int age;
     float height;
     float weight;
};
```

Classification of Member Functions

- **Accessor functions**
  - Allow retrieval of private/protected data members
- **Mutator functions**
  - Allow changing the value of a private/protected data members
- **Auxiliary functions**
  - Public if generally called outside function
  - Private/protected if only called by member functions

Different Access Scopes in C++

- **Access scope in C is simple**
  - Inside/outside a function

```
int g_total = 0;

int square(int x){
     return x * x;
}

int main()
{
    float num = 0;
    for (int i = 0; i < 10; i++){
        printf("the %d-th number is:", i);
        scanf("%d\n", &num);
        g_total += square(num);
    }
    return 0;
}
```

Global variable: can be accessed anywhere

Local variable: can be accessed within square()

Local variable: can be accessed within main()

Different Access Scopes in C++

- **Scope operator "::"**
  - Specify which scope a variable belongs to
  - Global scope

```
int g_total = 0;

int main()
{
    float num = 0;
    for (int i = 0; i < 10; i++)
    {
        printf("the %d-th number is:", i);
        scanf("%d\n", &num);
        ::g_total += num;
    }
        return 0;
}
```

Different Access Scopes in C++

- **Scope operator "::"**
  - Specify which scope a variable belongs to
  - Class scope

```
class person
{
public:
     void set_name(char* name);
     ...
      void print();
      ...
private:
     char* name;
     int age;
     float height;
     float weight;
};
```

```
void person::set_name(char* name)
{
     this->name = name;
}

void person::print()
{
     printf("the person's name: %s\n", name);
     printf("the person's age: %d\n", age);
     printf("the person's height: %4.1f\n", height);
     printf("the person's weight: %3.1f\n", weight);
}
```

Different Access Scopes in C++

- **Scope operator "::"**
  - Specify which scope a variable belongs to
  - *Static members* in a class
    - Static member variable: *all objects of the same class share the same properties*

```
lass vertex
{
public:
    float x, y, z;
    ...
    vertex();
private:
    static int object_count;
};
```

```
int vertex::object_count = 0;

vertex::vertex()
{
    x = y = z = 0.0f;
}
```

Initialization of static members should be global, outside of any member functions!

Different Access Scopes in C++

- **Scope operator "::"**
  - Specify which scope a variable belongs to
  - Static members in a class
    - Static member function

```
class vertex
{
public:
     float x, y, z;
     ...
     static int get_object_count();

     vertex();
     ~vertex();
private:
     static int object_count;
};
```

```
int vertex::object_count = 0;

vertex::vertex()
{
     x = y = z = 0.0f;
     object_count++;
}

vertex::~vertex()
{
     object_count--;
}

int vertex::get_object_count()
{
     return object_count;
}
```

Static member functions can only access static member variables

Different Access Scopes in C++

- **Scope operator "::"**
  - Specify which scope a variable belongs to
  - Static members in a class
    - What is the execution result?

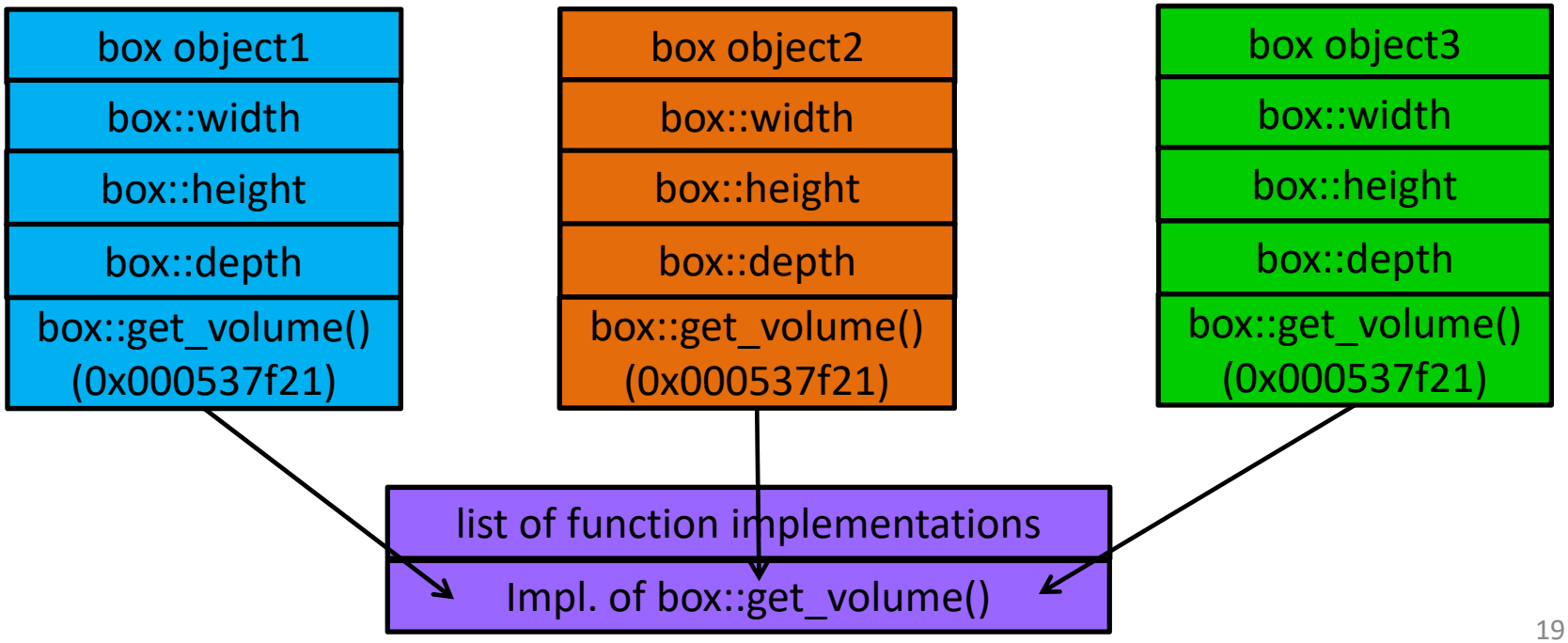
```
int main()
{
     vertex *p_v1=new vertex;
     vertex* p_v2 = new vertex;
     vertex* p_v3 = new vertex;

     printf("vertex object number: %d\n", p_v1->get_object_count());
     delete p_v1;
     printf("vertex object number: %d\n", p_v2->get_object_count());
     delete p_v2;
     printf("vertex object number: %d\n", p_v3->get_object_count());
     delete p_v3;

     return 0;
}
```

```
vertex object number: 3
vertex object number: 2
vertex object number: 1
```

How Object Data/Functions Are Arranged

```
class box
{
public:
     ...
     float get_volume();

     box(float, float, float);
private:
     float m_width, m_height, m_depth;
};
```

```
float box::get_volume()
{
     return m_width * m_height * m_depth;
}
```



Function Overloading

- **C++ allows you to specify more than one definition**
  - for a **function** name or an **operator** in the same scope
  - Must be *different in input arguments*
    - Different number of arguments
    - Different types of arguments
  - Only difference in return values will result in compilation error

Function Overloading

- **Why function overloading?**
  - Look at one simple example of adding numbers
    - How can we support different data types in C?

```
int add_int(int x, int y)
{
      return x + y;
}

long add_long(long x, long y)
{
      return x + y;
}
```

```
float add_float(float x, float y)
{
      return x + y;
}

double add_double(double x, double y)
{
      return x + y;
}
```

We should be careful on choosing the correct function!

Function Overloading

- **Why function overloading?**
  - Look at one simple example of adding numbers
    - With function loading in C++

```
int add(int x, int y)
{
      return x + y;
}

long add(long x, long y)
{
      return x + y;
}
```

```
float add(float x, float y)
{
      return x + y;
}

double add(double x, double y)
{
      return x + y;
}
```

You can ignore what specific "add" you should choose – the complier will choose for you

Function Overloading

- **Function overloading in a class**
  - Very useful for constructor
    - Look at previous example again

```
class person
{
public:
     ...
     person();
     person(const char* name, int age,
                  float height, float weight);
     person(const Person& p);
private:
     char* name;
     int age;
     float height;
     float weight;
};
```

Function Overloading

- **Function overloading in a class**
  - Any function overloading for destructor?
    - NO!!!
    - There is only one destructor

```
class line
{
public:
    char* get_string();

    line(const char* str);
    line(const line& str);
    ~line();
private:
    char* line_str;
    int size;
};
```

Function Overloading

- **Function overloading in a class**
  - Overloading for any other member functions

```
class vector
{
public:
     vector& add(const vector& v);
     vector& add(const float& v);
     vector& sub(const vector& v);
     vector& sub(const float& v);
     ...
     vector();
     vector(int dim);
     vector(const vector&);
     ~vector();
private:
     float* m_data;
     int m_dim;
};
```

```
vector& vector::add(const vector& v)
{
     for (long i = 0; i < m_dim; i++)
          m_data[i] += v.m_data[i];
     return (*this);
}

vector& vector::add(const float& v)
{
     for (long i = 0; i < m_dim; i++)
          m_data[i] += v;
     return (*this);
}
```

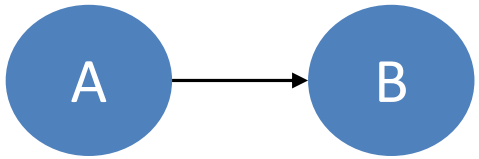
Class Inheritance

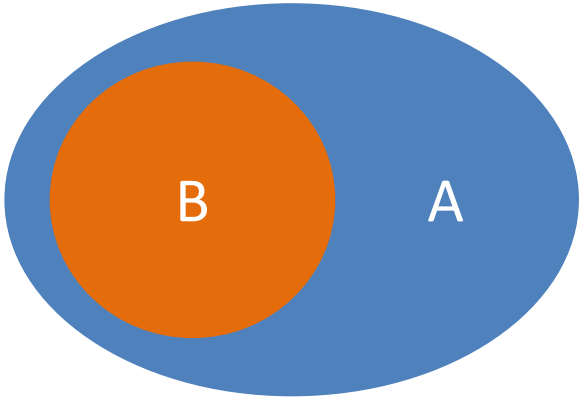
- **Think about code reuse?**
  - Important to successful coding
  - Efficient: no need to reinvent the wheel
  - Error reduction: code has been previously used/tested

- **What are the common ways?**
  - Functions
  - Classes
  - Class inheritance

Object Relationships

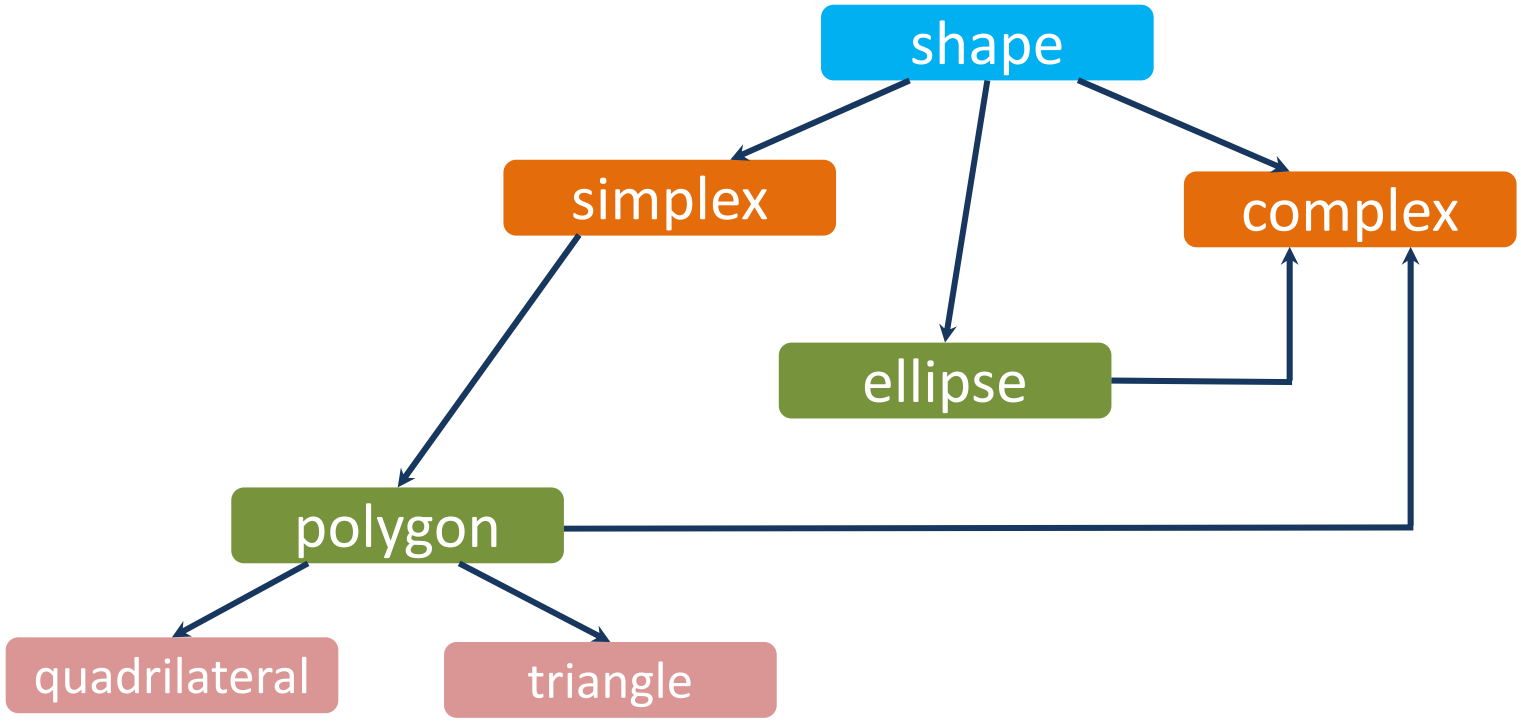
- **Two types of object relationships**
  - The **"is-a"** relationship
    - Inheritance
  - The **"has-a"** relationship
    - Composition
    - Aggregation





Object Relationships

- **We consider a representation of 2D shapes**



Object Relationships

- **Inheritance**

A simplex is a shape

- The simplex class is inherited from the shape class
- “shape” is the general class or the ***parent(base) class***
- “simplex” is the specialized class, or ***child(derived) class***, that inherits from “shape”

Class Inheritance

- **Definition of shape class**
  - "shape" class is a general class defining the shared functionalities among all shapes

```
class shape2D
{
public:
    float get_boundary_length();
    float get_area();

    shape2D();
protected:
    float m_boundary_length;
    float m_area;
};
```

protected specifier allows being accessed by derived classes

Class Inheritance

- **Implementation of shape class**

```
shape2D::shape2D()
{
    m_boundary_length = 0.0f;
    m_area = 0.0f;
}

float shape2D::get_boundary_length()
{
    return m_boundary_length;
}

float shape2D::get_area()
{
    return m_area;
}
```

Class Inheritance

- **Definition of simplex class**
  - "simplex" class is also a more general class, which is derived from the shape class
  - But it has more specific definitions

```
class simplex2D : public shape2D
{
public:
    int get_vertex_count();
    int get_edge_count();

    simplex2D();
protected:
    int m_vertex_count;
    int m_edge_count;
};
```

Parent(base) class

Inheritance access specifier

Class Inheritance

- **Implementation of simplex class**
  - Note that by inheritance, defined functions in "shape" class no longer need implementation

```
simplex2D::simplex2D()
{
     m_vertex_count = 0;
     m_edge_count = 0;
}

int simplex2D::get_vertex_count()
{
      return m_vertex_count;
}

int simplex2D::get_edge_count()
{
      return m_edge_count;
}
```

Class Inheritance

- **Use of child(derived) class object**
  - Can use the feature both in child(derived) and parent(base) classes

```
int main()
{
    simplex2D sim;

    printf("the area of the shape:%f\n",
                sim.get_area());
    printf("the vertex number of the shape:%d\n",
                sim.get_vertex_count());

    return 0;
}
```

Call the parent class function

Call the child class function

Object Relationships

- **Class composition**
  - Think about if we further derive a polygon class from the base class simplex
  - It requires the vertex representation – another class as a member variable

A polygon uses vertices

- The polygon class and its further derived classes use vertex class to represent their own vertices
- These classes compose of vertex class objects

Class Composition

- **Definition of vertex class**
  - It will be used in polygon and its derived classes

```
class vertex2D
{
public:
    float x, y;

    vertex2D& assign (const vertex2D&);

    vertex2D();
    vertex2D(float, float);
    vertex2D(const vertex2D&);
};
```

Class Composition

- **Implementation of vertex class**

```
vertex2D::vertex2D()
{
    x = y = 0.0f;
}

vertex2D::vertex2D
        (float x, float y)
{
    this->x = x;
    this->y = y;
}
```

```
vertex2D::vertex2D
        (const vertex2D& v)
{
    x = v.x;
    y = v.y;
}

vertex2D& vertex2D::assign
        (const vertex2D& v)
{
    x = v.x;
    y = v.y;

    return (*this);
}
```

Class Composition

- **Definition of polygon class**
  - Derived from simplex class and uses vertex class

```
class polygon2D : public simplex2D
{
public:
    void calc_boundary_length();

    polygon2D();
    polygon2D(vertex2D* p_vertex, int vertex_count);
    ~polygon2D();
protected:
    vertex2D* m_vertex;
};
```

Class Composition

- **Implementation of polygon class**

```
polygon2D::polygon2D()
{
    m_vertex = NULL;
}

polygon2D::polygon2D(vertex2D* p_vertex, int vertex_count)
{
    m_vertex = new vertex2D[vertex_count];
    memcpy(m_vertex, p_vertex, sizeof(vertex2D));
    m_vertex_count = vertex_count;
    m_edge_count = vertex_count - 1;
}

polygon2D::~polygon2D()
{
    if (m_vertex != NULL)
        delete []m_vertex;
}
```

Class Composition

- **Implementation of polygon class**

```
void polygon2D::calc_boundary_length()
{
    float total_len = 0.0f;
    for (int i = 0; i < m_vertex_count - 1; i++)
    {
        float delta_x = m_vertex[i + 1].x - m_vertex[i].x;
        float delta_y = m_vertex[i + 1].y - m_vertex[i].y;
        total_len += (float)sqrt(delta_x * delta_x
                                 + delta_y * delta_y);
    }
    m_boundary_length = total_len;
}
```

Class Composition

- **Use of polygon class**

```
int main()
{
    vertex2D vertex[5];
    for (int i=0; i < 5; i++)
    {
        vertex[i].x = 10 * float(rand()) / RAND_MAX;
        vertex[i].y = 10 * float(rand()) / RAND_MAX;
    }

    polygon2D p(vertex, 5);
    p.calc_boundary_length();

    printf("the boundary length of polygon is: %f\n",
                                    p.get_boundary_length());

    return 0;
}
```

Further Class Inheritance and Composition

- **Definition of triangle class**
  - Triangle class is considered as a special case of polygon: reply on polygon representation

```
class triangle2D : public polygon2D
{
public:
    void calc_area();

    triangle2D();
    triangle2D(const vertex2D&,
               const vertex2D&, const vertex2D&);
};
```

Further Class Inheritance and Composition

- **Implementation of triangle class**

```
triangle2D::triangle2D(){

}

triangle2D::triangle2D(const vertex2D& v1,
                       const vertex2D& v2, const vertex2D& v3){
     m_vertex = new vertex2D[3];
     m_vertex[0] = v1;
     m_vertex[1] = v2;
     m_vertex[2] = v3;
     m_vertex_count = 3;
     m_edge_count = 2;
}

void triangle2D::calc_area(){
     vertex2D& A = m_vertex[0];
     vertex2D& B = m_vertex[1];
     vertex2D& C = m_vertex[2];

     m_area = (float)fabs((A.x * (B.y - C.y) +
          B.x * (C.y - A.y) + C.x * (A.y - B.y)) / 2.0f);
}
```

Further Class Inheritance and Composition

- **Use of triangle class**

```
int main()
{
    vertex2D v1, v2, v3;
    v1 = vertex2D(-1.3, -5.6);
    v2 = vertex2D(3.7, 0.9);
    v3 = vertex2D(0.2, 4.7);

    triangle2D t(v1, v2, v3);

    t.calc_boundary_length();
    t.calc_area();

    printf("the boundary length of triangle is: %f\n",
                t.get_boundary_length());
    printf("the area of triangle is: %f\n", t.get_area());

    return 0;
}
```

Call parent polygon's member

Call triangle's new member

Call triangle's (root) parent members

Class Aggregation

- **Definition of complex class**
  - Aggregate a set of different simplex classes

```
class ellipse2D :
          public shape2D
{
public:
     void set_radius
          (float, float);

     ellipse2D();
     ellipse2D
          (float, float);
protected:
     float m_r1, m_r2;
};
```

```
class complex : public shape2D
{
public:
     void calc_boundary_length();
     void calc_area();

     void set_polygon(polygon2D*, int);
     void set_ellipse(ellipse2D*, int);

     complex();
     ~complex();
protected:
     polygon2D* m_poly;
     int m_poly_count;

     ellipse2D* ellipse;
     int m_ellipse_count;
};
```

Inheritance Access Specifiers

- **Inheritance can be specified with different access specifiers**
  - public:
    - All the access properties in the parent class is not changed
    - Private members will not be accessed in child classes

```
class simplex2D : public shape2D
{
public:
     int get_vertex_count();
     int get_edge_count();

     simplex2D();
protected:
     int m_vertex_count;
     int m_edge_count;
};
```

```
int main()
{
     simplex2D sim;

     float len=sim.get_boundary_length();

     int vertex_num = sim.get_vertex_count();
     ...

     return 0;
}
```

Inheritance Access Specifiers

- **Inheritance can be specified with different access specifiers**
  - protected:
    - All the public members will be changed to protected
    - Other members are not affected

```
class simplex2D : protected shape2D
{
public:
     int get_vertex_count();
     int get_edge_count();

     simplex2D();
protected:
     int m_vertex_count;
     int m_edge_count;
};
```

```
int main()
{
     simplex2D sim;

     float len=sim.get_boundary_length();

     int vertex_num = sim.get_vertex_count();
     ...

     return 0;
}
```

Inheritance Access Specifiers

- **Inheritance can be specified with different access specifiers**
  - private:
    - All the members will be private
    - Derived classes cannot access any member in parent class

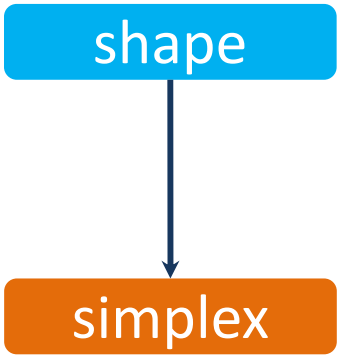
```
class polygon2D : public simplex2D
{
public:
     void calc_boundary_length();

     polygon2D();
     polygon2D(vertex2D* p_vertex,
                    int vertex_count);
     ~polygon2D();
protected:
     vertex2D* m_vertex;
};
```

```
void polygon2D::calc_boundary_length(){
     float total_len = 0.0f;
     for (int i = 0; i < m_vertex_count - 1; i++){
          float delta_x =
                    m_vertex[i + 1].x - m_vertex[i].x;
          float delta_y =
                    m_vertex[i + 1].y - m_vertex[i].y;
          total_len += (float)sqrt(delta_x * delta_x +
                                   delta_y * delta_y);
     }
     m_boundary_length = total_len;
}
```

Private member in base class “shape2D”

Calling Constructors Across Class Hierarchies

- **How constructors are called over class hierarchies?**
  - Parent class constructors will first be called before calling child class constructors



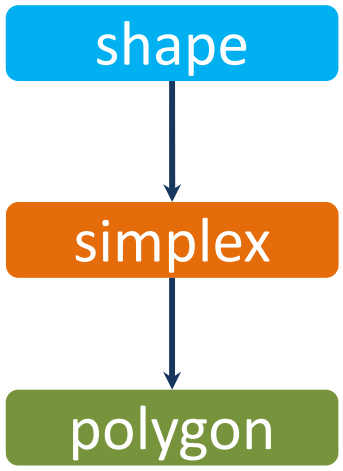

```
simplex2D sim;
```

```
shape2D::shape2D()
{
      m_boundary_length = 0.0f;
      m_area = 0.0f;
}
```

```
simplex2D::simplex2D()
{
      m_vertex_count = 0;
      m_edge_count = 0;
}
```

Calling Constructors Across Class Hierarchies

- **How constructors are called over class hierarchies?**
  - Parent class constructors will first be called before calling child class constructors




```
shape2D::shape2D()
{
    m_boundary_length = 0.0f;
    m_area = 0.0f;
}
```

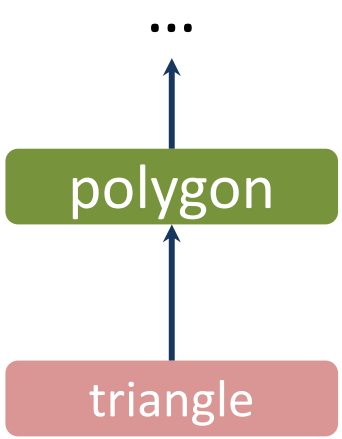
```
simplex2D::simplex2D()
{
    m_vertex_count = 0;
    m_edge_count = 0;
}
```

```
vertex2D v[5];
...
polygon2D p(v, 5);
```

```
polygon2D::polygon2D(vertex2D* p_vertex,
                     int vertex_count)
{
    m_vertex = new vertex2D[vertex_count];
    memcpy(m_vertex,
        p_vertex, sizeof(vertex2D));
    m_vertex_count = vertex_count;
    m_edge_count = vertex_count-1;
}
```

Calling Destructors Across Class Hierarchies

- **How destructors are called over class hierarchies?**
  - Reverse order:
    - Called from child classes to parent classes




```
...
polygon2D::~polygon2D()
{
     if (m_vertex != NULL)
          delete []m_vertex;
}


triangle2D::~triangle2D()
{

}
```

Overloading v.s. Overriding

- **Overloading**
  - Use the same function name, but with different parameters for each overloaded implementation
- **Overriding**
  - Use the same function name and parameters, but with a different implementation ***cross class hierarchies***
  - Child class method “hides” parent class method
  - Only possible by using inheritance

Function Overriding over Class Hierarchies

- **Each child class can have its own version of the same function**
  - The function is called corresponding to the class type of the object
  - E.g., return back to our shape example

```
class polygon2D : public simplex2D
{
public:
     void calc_boundary_length();
     float get_area();
     ...
protected:
     vertex2D* m_vertex;
};
```

```
class triangle2D : public polygon2D
{
public:
     float get_area();
     ...
};
```

Function Overriding over Class Hierarchies

- **Each child class can have its own version of the same function**
  - E.g., return back to our shape example

```
float polygon2D::get_area(){
     float total_area = 0;
     for (int i = 1; i < m_vertex_count - 1; i++){
          vertex2D& A = m_vertex[0];
          vertex2D& B = m_vertex[i];
          vertex2D& C = m_vertex[i+1];

          total_area+= (float)fabs((A.x * (B.y - C.y) +
              B.x * (C.y - A.y) + C.x * (A.y - B.y)) / 2.0f);
     }

     m_area = total_area;

     return m_area;
}
```

```
float triangle2D::get_area()
{
     calc_area();
     return polygon2D::get_area();
}
```

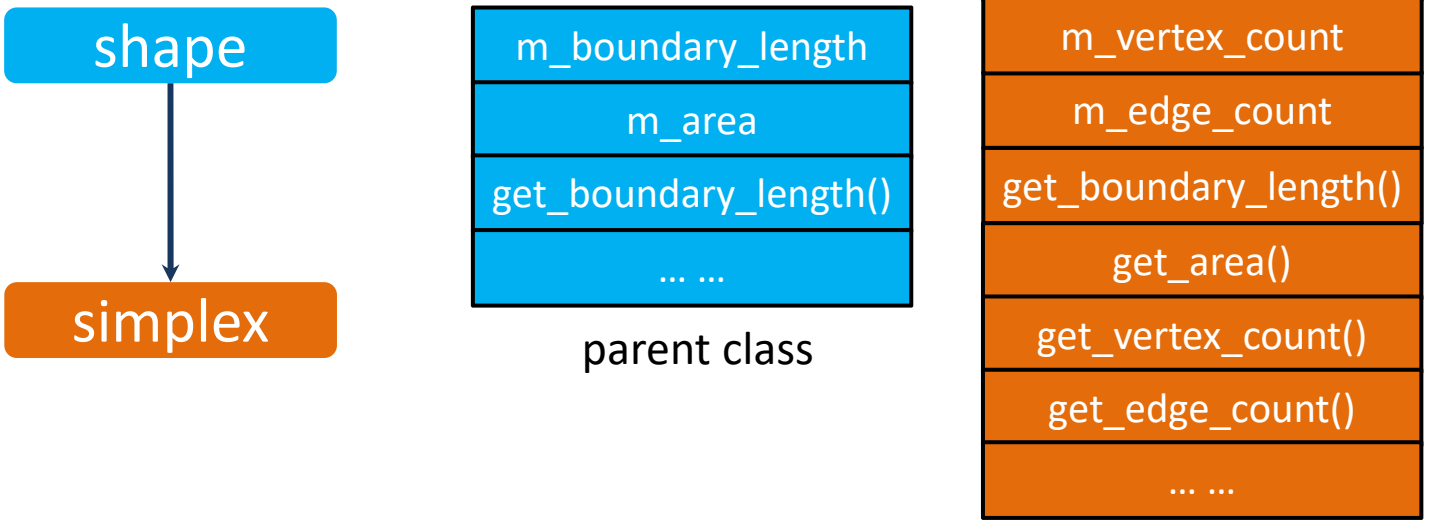
Call overridden function directly from base class

Access from Child Class to Parent Class

- **Two types of member function access**
  - *Non-overridden function*
    - Directly called without any scope specification
  - *Overridden function*
    - Ambiguity without scope specification
    - Should specify which base class the overridden function is being called

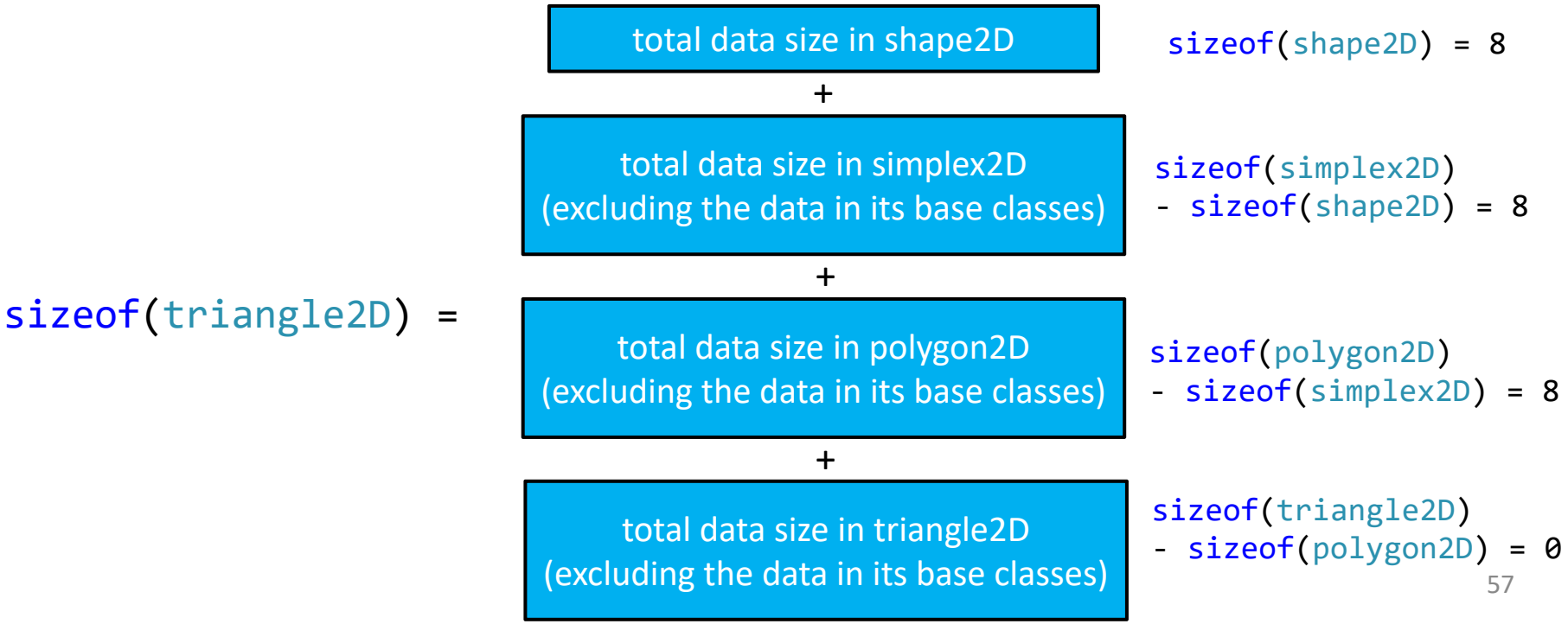
Data Arrangement for Child Classes

- **The child class will contain data/function addresses from parent classes**
  - Data from parent classes are arranged in memory before child classes



Size of an Object from a Derived Class

- **What is the size of a child class object?**
  - The size of all parent classes
  - Plus the size of itself



Actual Size of a Class Object

- **Look at how to we compute the size of a structure**
  - Previously we introduce the following

```
struct A
{
    int a;
    char b;
};
```

```
sizeof(A) = sizeof(int) + sizeof(char) = 5
```

Is this true in practice? No!

Actual Size of a Class Object

- **Data padding**
  - Compiler will pad the data for copy efficiency

```
In practice:

sizeof(A) = sizeof(int) + sizeof(int) = 8
```

  - Padding is tricky
    - Always use sizeof() to compute the size of a class or structure

Variable Ambiguity in Inheritance

- **What if parent and child classes have member variables of the same name?**
  - Use scope operator to differentiate

```
class simplex2D : public shape2D
{
public:
     int get_vertex_count();
     int get_edge_count();

     simplex2D();
protected:
     int m_vertex_count;
     int m_edge_count;
};
```

```
class polygon2D : public simplex2D
{
public:
     void calc_boundary_length();

     polygon2D();
     polygon2D(vertex2D* p_vertex,
                         int vertex_count);
     ~polygon2D();
protected:
     vertex2D* m_vertex;
     int m_vertex_count;
     int m_edge_count;
};
```

Variable Ambiguity in Inheritance

- **What if parent and child classes have member variables of the same name?**
  - Use scope operator to differentiate

```
triangle2D::triangle2D(const vertex2D& v1,
                       const vertex2D& v2, const vertex2D& v3)
{
     m_vertex = new vertex2D[3];
     m_vertex[0] = v1;
     m_vertex[1] = v2;
     m_vertex[2] = v3;
     m_vertex_count = 3;
     m_edge_count = 2;

     simplex2D::m_vertex_count = m_vertex_count;
     simplex2D::m_edge_count = m_edge_count;
}
```

Variable Ambiguity in Inheritance

- **What is the function parameter has the same name as the class member?**
  - Use “this” pointer to differentiate

```
class polygon2D : public simplex2D
{
public:
     void calc_boundary_length();

     polygon2D();
     polygon2D(vertex2D* vertex,
                    int vertex_count);
     ~polygon2D();
protected:
     vertex2D* vertex;
     int vertex_count;
     int edge_count;
};
```

```
polygon2D::polygon2D
     (vertex2D* vertex, int vertex_count)
{
     this->vertex =
          new vertex2D[vertex_count];
     memcpy(this->vertex,
           vertex, sizeof(vertex2D));
     this->vertex_count = vertex_count;
     this->edge_count = vertex_count-1;
}
```

Type Conversion Between Classes

- **Type conversion can be performed between child and parent classes**
  - Safe conversion: from child to parent
  - Child data/functions will be discarded

```
vertex2D v1, v2, v3;
...

triangle2D t(v1, v2, v3);
t.calc_area();

polygon2D p = (polygon2D)t;
p.calc_boundary_length();

simplex2D s = (simplex2D)p;

printf("the boundary length: %f\n", s.get_boundary_length());
printf("the shape area: %f\n", s.get_area());
printf("the vertex count: %d\n", s.get_vertex_count());
printf("the edge count: %d\n", s.get_edge_count());
```

Call triangle's member function

Call polygon's member function

Call simplex's member function

Type Conversion Between Classes

- **Type conversion can be performed between child and parent classes**
  - Safe conversion: from child to parent
  - Child data/functions will be discarded

  - False case (compilation error)

```
shape2D shape;
triangle2D triangle = (triangle2D)shape;
```

Type Conversion Between Classes

- **Type conversion can be performed between child and parent classes**
  - Use class pointers (*more tricky*)
    - Safe conversion: from child to parent classes

```
vertex2D v1, v2, v3;
...
triangle2D *p_t=new triangle2D(v1, v2, v3);
p_t->calc_area();

polygon2D *p_p = (polygon2D*)p_t;
p_p->calc_boundary_length();

simplex2D *p_s = (simplex2D*)p_t;
//simplex2D* p_s = (simplex2D*)p_p;

printf("the boundary length: %f\n", p_s->get_boundary_length());
printf("the shape area: %f\n", p_s->get_area());
printf("the vertex count: %d\n", p_s->get_vertex_count());
printf("the edge count: %d\n", p_s->get_edge_count());
... //don't forget to destroy triangle pointer
```

Type Conversion Between Classes

- **Type conversion can be performed between child and parent classes**
  - Use class pointers (*should be more careful*)
    - Safe conversion: from child to parent classes
    - What if the following code is written?

```
shape2D shape;
triangle2D* p_triangle = (triangle2D*)&shape;
p_triangle->calc_area();
```

No compilation error!
But logically wrong! Will crash!!!