C++ Class Basics

한양대학교 컴퓨터소프트웨어학부

2017년 2학기

Structured Information

```
Name     ID       Grade    Midterm  Final    HW1      HW2
-------- -------- -------- -------- -------- -------- --------
gdhong   13001    A+       99       90       85       100
cskim    13002    A        80       95       93       90
yhlee    13003    B+       85       80       92       88
...
```

- Informations of students taking a class.
  - Name, id number, grade, scores of exams and homeworks.

- How are you going to represent and process these data?

Bunch of Arrays

- One option : use arrays.

```
Name     ID       Grade    Midterm  Final    HW1      HW2
-------- -------- -------- -------- -------- -------- --------
gdhong   13001    A+       99       90       85       100
cskim    13002    A        80       95       93       90
yhlee    13003    B+       85       80       92       88
...
```

- Problems?

```
void ProcessGrade(int num_students, const string* names, const string* ids,
                  const int* midterm, const int* final,
                  const int* hw1, const int* hw2, string* grades) {
  for (int i = 0; i < num_students; ++i) {
    int sum = midterm[i] + final[i] + hw1[i] + hw2[i];
    if (sum/4 >= 95) grades[i] = "A+";
    else if (sum/4 >= 90) grades[i] = "A";
    else if (sum/4 >= 85) grades[i] = "B+";
    ...
  }
}
```

Bunch of Arrays

```
Name     ID       Grade    Midterm  Final    HW1      HW2
-------- -------- -------- -------- -------- -------- --------
gdhong   13001    A+       99       90       85       100
cskim    13002    A        80       95       93       90
yhlee    13003    B+       85       80       92       88
...
```

Some of problems:

- What if HW3 is added?
- How are all arrays allocated, initialized, and deallocated?
- How is it guaranteed that all arrays have the same size?

```
void ProcessGrade(int num_students, const string* names, const string* ids,
                  const int* midterm, const int* final,
                  const int* hw1, const int* hw2, string* grades);
…
int main() {
  int num_students = 30;
  string names = new string[num_students];
  ...
  int midterm = new int[num_students];
  int final = new int[num_students];
  int hw1 = new int[num_students];
  ...
}
```

C/C++ Struct

```
Name     ID       Grade    Midterm  Final    HW1      HW2
-------- -------- -------- -------- -------- -------- --------
gdhong   13001    A+       99       90       85       100
cskim    13002    A        80       95       93       90
yhlee    13003    B+       85       80       92       88
...
```

- Another option : use structs.

```
struct Student {
  string name, id, grade;
  int midterm, final, hw1, hw2;
};

void ProcessGrade(Student* students, int num_students) {
  for (int i = 0; i < num_students, ++i) {
    int sum = students[i].midterm + students[i].final +
        students[i].hw1 + students[i].hw2;
    if (sum/4 >= 95) students[i].grade = "A+";
    else if (sum/4 >= 90) students[i].grade = "A";
    else if (sum/4 >= 85) students[i].grade = "B+";
    ...
  }
  ...
}
```

C/C++ Struct

```
struct Student {
  string name, id, grade;
  int midterm, final, hw1, hw2;
};
```

C/C++ struct

- User-defined type representing a structured record.
- `struct` is useful for packaging the related information, and passing the packaged info around.
- Use ‘`.`’ to access a field in a structure.
  When using a pointer, use ‘`->`’ (`p->a` is same as `(*p).a`).

```
Student
  name:      gdhong
  id:        13001
  grade:     A+
  midterm:   99
  final:     90
  hw1:       85
  hw2:       100
```

```
Student
  name:      cskim
  id:        13002
  grade:     A
  midterm:   80
  final:     95
  hw1:       93
  hw2:       90
```

```
Student
  name:      yhlee
  id:        13003
  grade:     B+
  midterm:   85
  final:     80
  hw1:       92
  hw2:       88
```

Array of Structs

```
Name     ID       Grade    Midterm  Final    HW1      HW2
-------- -------- -------- -------- -------- -------- --------
gdhong   13001    A+       99       90       85       100
cskim    13002    A        80       95       93       90
yhlee    13003    B+       85       80       92       88
...
```

- Array of structs.

```
struct Student {
  string name, id, grade;
  int midterm, final, hw1, hw2;
};

void ProcessGrade(Student* students, int num_students);

int main() {
  int num_students = 30;
  Student* students = new Student[num_students];
  // Initialize students array.
  ...
  ProcessGrade(students, num_students);
  delete[] students;
}
```

Structure Initialization

- Initializing individual fields or initializing as a whole :

```
struct Student {
  string name, id, grade;
  int midterm, final, hw1, hw2;
};

int main() {
  int num_students = 30;
  Student* students = new Student[num_students];
  for (int i = 0; i < num_students; ++i) {
    cin >> students[i].name;  // Use . to access the field.
    cin >> (students + i)->id;  // For a pointer, use -> instead.
    cin >> students[i].midterm >> students[i].final
        >> students[i].hw1 >> students[i].hw2;
  }

  Student a_student = { "gdhong", "13001", "", 99, 90, 85, 100 };  // OK.
  students[0] = { "gdhong", "13001", "", 99, 90, 85, 100 };  // Compile error.
  ...
}
```

User-defined Type

- A structure can be considered as a user-defined type.
  - In C++, the struct name can be used just like a type name.
  - In C, it should either used as ‘`struct Name`’, or do `typedef`.

```
// C example.

struct Student {
  string name, id, grade;
  int midterm, final, hw1, hw2;
};

typedef struct Student StudentType;

void ProcessGrade(struct Student* students, int num_students);

int main() {
  int num_students = 30;
  StudentType students = new StudentType[num_students];
  ...
}
```

Information Hiding

- All fields in a struct is ‘visible’ to the users.
  - Users can read the information in the fields and also can modify them without any restriction.
  - It can break the integrity of the information in the structure.

```
struct Student {
  string name, id, grade;
  int midterm, final, hw1, hw2;
};

// Use this function to compute the grade.
void ProcessGrade(Student* students, int num_students);

int main() {
  Student a_student = { "gdhong", "13001", "", 99, 90, 85, 100 };

  ProcessGrade(&a_student, 1);  // The grade for "gdhong" is computed.

  a_student.grade = "D-";  // But it is updated incorrectly here.
  a_student.grade = "hello_world";  // Or it may have any arbitrary string.
  ...
}
```

Information Hiding in C++ Classes

- Classes are very similar to structs, except the access control.
  - The fields are either `public`, `private`, or `protected`.
  - `public` fields are accessible by everyone.
  - `private` fields are only accessible by its member functions*.
  - `protected` fields are accessible by its member functions and its successors**.
- In other words, structs are the classes whose fields are all public.
- What are the ‘**member functions**’?
  - The data fields in structs or classes are called as **member variables**.

Class Member Functions

- Classes can have member functions.
  - Member functions are declared in the class definition.
  - Member functions are defined either in the class definition or outside of the class definition.
  - To use member functions, use just as a fields and a function.

```
struct Student {
  string name, id, grade;
  int midterm, final, hw1, hw2;

  void ProcessGrade();  // Declare ProcessGrade member function.
};

// Define the member function here.
void Student::ProcessGrade() {
  ...
}

int main() {
  Student a_student = { "gdhong", "13001", "", 99, 90, 85, 100 };
  a_student.ProcessGrade();  // Call the member function ProcessGrade.
  ...
}
```

Class Member Functions

- The member functions can access the member variables.

```
struct Student {
  string name, id, grade;
  int midterm, final, hw1, hw2;

  void ProcessGrade();  // Declare ProcessGrade member function.
};

// Define the member function here.
void Student::ProcessGrade() {
  int sum = midterm + final + hw1 + hw2;
  if (sum/4 >= 95) grade = "A+";
  else if (sum/4 >= 90) grade = "A";
  else if (sum/4 >= 85) grade = "B+";
    ...
}

int main() {
  Student a_student = { "gdhong", "13001", "", 99, 90, 85, 100 };
  a_student.ProcessGrade();  // Call the member function ProcessGrade.
  ...
}
```

Class Member Functions

- Let’s try a class instead of a struct.

```
class Student {
 public:
  void SetInfo(string name, string id) { name_ = name, id_ = id; }
  void SetScores(int midterm, int final, int hw1, int hw2) {
    midterm_ = midterm, final_ = final, hw1_ = hw1, hw2_ = hw2;
  }
  void ProcessGrade() { ... }
  string GetGrade() { return grade_; }

 private:
  string name_, id_, grade_;
  int midterm_, final_, hw1_, hw2_;
};

int main() {
  Student a_student;
  a_student.SetInfo("gdhong", "13001");
  a_student.SetScores(99, 90, 85, 100);
  a_student.ProcessGrade();  // Call the member function ProcessGrade.

  a_student.grade_ = "D-";  // Compile error!
  string grade = a_student.GetGrade();  // Fine.
  ...
}
```

`this` - Pointer to the Instance

- In member functions, `this` can be used to point the instance itself.

```
class Student {
 public:
  void SetInfo(string name, string id) { this->name_ = name, id_ = id; }

  void SetScores(int midterm, int final, int hw1, int hw2) {
    midterm_ = midterm, final_ = final, hw1_ = hw1, hw2_ = hw2;
    this->ProcessGrade();
  }

  void ProcessGrade() { ... }

  string GetGrade() { return grade_; }

 private:
  string name_, id_, grade_;
  int midterm_, final_, hw1_, hw2_;
};
```

Basic Class Design

- Hide all data members, unless it is absolutely required (hardly it is).
  - Make accessors and setters if necessary.
  - Name member variables differently to distinguish them from local variables in member functions
    (e.g. `name_`).
- Make member functions meaningful and atomic.
  - Name member functions appropriately and write a detailed comment near the declarations.
  - Users must be able to understand what the member function does without reading its function definition.
- Coding style guide:
  - Variables : lower-case letters and '_'
  - Class and function names : CamelCase

C/C++ Const

`const` : the instance remains constant during the operation / life.

```
void TestConst(int a, const int b,
               char* p, const char* cp) {
  int i = a, j = b;  // Both OK.
  a = i * 2;         // OK.
  b = i + j;         // Error: assignment of read-only location

  p[0] = 'a';        // OK;
  cp[0] = 'b';       // Error: assignment of read-only location

  char* q = NULL;
  const char* cq = "hello";
  p = q, cp = q;     // Both OK.
  p = cq;            // Warning: assignment discards qualifiers from
                     //          pointer target type.
  cp = cq;           // OK.
}
```

Const Member Function

A member function can be const if it does not change any data members.

```
struct Student {
 public:
  // These three functions are not const.
  void SetInfo(string name, string id) { name_ = name, id_ = id; }
  void SetScores(int midterm, int final, int hw1, int hw2) {
    midterm_ = midterm, final_ = final, hw1_ = hw1, hw2_ = hw2;
  }
  void ProcessGrade() { ... }
  // This function is const since it does not change any members.
  string GetGrade() const { return grade_; }

 private:
  string name_, id_, grade_;
  int midterm_, final_, hw1_, hw2_;
};
```

- Give the information (that the class instance will remain unchanged) to the compiler and the class user.

C++ Reference (&)

Reference data type : think of it as a referenced pointer.

- Less powerful but safer than the pointer type.
- Must be initialized at the creation.
- The association cannot be changed later.

```
int a = 10;
int& b = a;  // b is an alias of a.
b = 20;
assert(a == 20 && b = 20);

int* p = &a;
*p = 30;
assert(a == 30 && *p == 30);

int& bb;  // Error: ‘bb’ declared as reference but not initialized

const int& c = a;
c = 10;  // Error: assignment of read-only reference ‘c’
a = 10;  // OK.
assert(a == 10 && c == 10);
```

C++ Reference (&)

- Remember C/C++ parameter passing and return copies the data.
  - Use pointers or reference to avoid this.
- Passing arguments using reference type (&)
  - Avoids copying the arguments.
  - Guarantees reference to a valid instance.
  - The instances may be modified by the function.

```
struct Triplet { int a, b, c; };

void TestReference(Triplet t, Triplet* pt, Triplet& rt) {
  t.a = 10, pt->b = 20, rt.c = 30;
}

int main() {
  Triplet triplet;
  triplet.a = 0, triplet.b = 0, triplet.c = 0;

  TestReference(triplet, &triplet, triplet);
  assert(triplet.a == 0 && triplet.b == 20 && triplet.c == 30);

  TestReference(triplet, NULL, triplet);  // Causes SEGFAULT.
  return 0;
}
```

a	10
b	0
c	0

a	0
b	20
c	30

C++ Const Reference (const &)

- Passing arguments using const reference type (const &)
  - Avoids copying the arguments.
  - Guarantees reference to a valid instance.
  - The instances remains unchanged after the function call.

```
struct Triplet { int a, b, c; };

void TestConstReference(const Triplet ct, const Triplet* cpt,
                        const Triplet& crt) {
  ct.a = 10, cpt->b = 20, crt.c = 30;  // All are errors.
  printf("%d, %d, %d\n", ct.a, cpt->b, crt.c);
}

int main() {
  Triplet triplet;
  triplet.a = 10, triplet.b = 20, triplet.c = 30;

  TestConstReference(triplet, NULL, triplet);  // Causes SEGFAULT.
  return 0;
}
```

Basic Class Design 2

- Hide all data members, unless it is absolutely required (hardly it is).
- Make member functions meaningful and atomic.

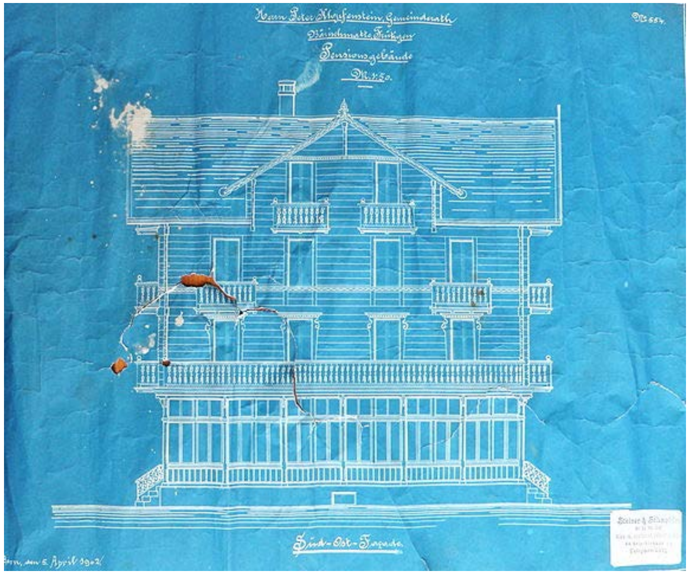
- Use `const` as much as possible.
  - If a member function is (conceptually) const, make it const.
  - If a local variable is unchanged, make it const.
- Use (const) reference or pointers in function parameters,
  especially when passing a class instance.

- Coding style guide: make in and out parameter clearly visible.
  - Order - input parameters then output (mixed) parameters.
  - Type - use const reference (const &) for input parameters, and
    pointers for output parameters; const pointer can be used when it can be NULL.

Class Instantiation

Classes vs. Instances

- Analogous to blueprints vs. buildings.
- Instantiation : making an instance of the class/type.
    - Instances have allocated memory to store specific info.
    - There can be multiple identical instances of the same type, but there cannot exist identical types/classes.



Constructor and Destructor

For any class instance (either dynamically allocated, local, or member),

- Constructor : when it is created, setup necessary stuffs.
- Destructor : when it is destroyed (freed), clean up the stuffs.

Class Constructor

- Constructors are special member functions that are used to initialize the object.
- They have the same name as the class and no return type, but may have different arguments.
- Use '`: field(value), ...`' to initialize the member variables.

```
class Student {
 public:
  Student() : name_(), id_(), grade_(),
      midterm_(0), final_(0), hw1_(0), hw2_(0) {}
  Student(const string& name, const string& id) : name_(name), id_(id) {
    midterm_ = 0, final_ = 0, hw1_ = 0, hw2_ = 0;
  }

  void SetInfo(const string& name, const string& id) {
    name_ = name, id_ = id;
  }
  const string& grade() const { return grade_; }
  ...

 private:
  string name_, id_, grade_;
  int midterm_, final_, hw1_, hw2_;
};
```

Class Destructor

- The destructor is a special member function for clean-up that is called when the object is destructed.
- Its name is '~' + the class name.
- It has no arguments and no return type.

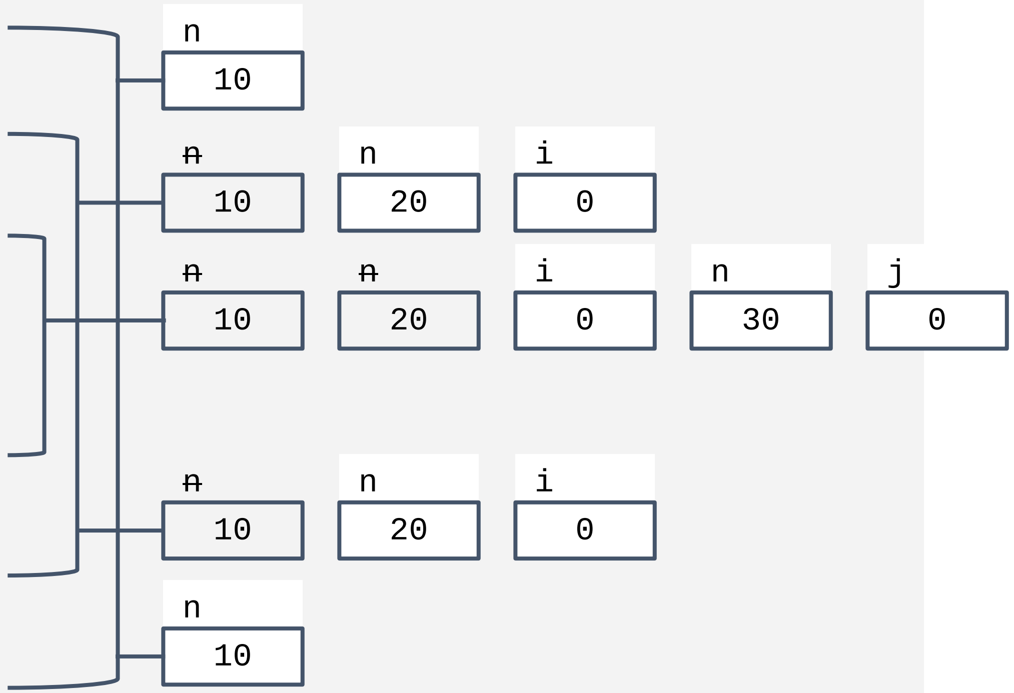
```
class Student {
 public:
  Student() { midterm_ = 0, final_ = 0, hw1_ = 0, hw2_ = 0; }
  Student(const string& name, const string& id) {
    SetInfo(name, id);
    midterm_ =0, final_ = 0, hw1_ = 0, hw2_ = 0;
  }
  ~Student() { /* Nothing to do. */ }

  void SetInfo(const string& name, const string& id) { ... }
  const string& grade() const { return grade_; }
  ...

 private:
  string name_, id_, grade_;
  int midterm_, final_, hw1_, hw2_;
};
```

Constructor / Destructor Example

```
class DoubleArray {
 public:
  DoubleArray() : ptr_(NULL), size_(0) {}
  DoubleArray(size_t size) : ptr_(NULL), size_(0) { Resize(size); }

  ~DoubleArray() { if (ptr_) delete[] ptr_; }

  void Resize(size_t size);

  int size() const { return size_; }
  double* ptr() { return ptr_; }
  const double* ptr() const { return ptr_; }

 private:
  double* ptr_;
  size_t size_;  // size_t is unsigned int.
};

void DoubleArray::Resize(size_t size) {
  double* new_ptr = new double[size];
  if (ptr_) {
    for (int i = 0; i < size_ && i < size; ++i) new_ptr[i] = ptr_[i];
    delete[] ptr_;
  }
  ptr_ = new_ptr;
  size_ = size;
}
```

C/C++ Scope Example

```
void TestScope(int n) {
  assert(n == 10);

  for (int i = 0; i < n; ++i) {
    int n = 20;

    for (int j = 0; j < n; ++j) {
      int n = 30;

      ...

      assert(n == 30);
    }
    // Note j is out of scope.
    assert(n == 20);
  }
  // Note i is out of scope.
  assert(n == 10);
}

int main() {
  TestScope(10);
  return 0;
}
```



Scope and Constructor / Destructor

```
struct TestClass {
  int n;
  TestClass(int i) : n(i) { cout << "Constructor " << n << endl; }
  ~TestClass() { cout << "Destructor " << n << endl; }
};

void TestClassScope(int n) {
  TestClass c1(n);
  for (int i = 0; i < n; ++i) {
    TestClass c2(i);
  }
}

int main() {
  TestClassScope(3);
  return 0;
}
```

```
Constructor 3
Constructor 0
Destructor 0
Constructor 1
Destructor 1
Constructor 2
Destructor 2
Destructor 3
```

C Structure Example : Complex Number

```
struct Complex {
  double real;
  double imag;
};

int main() {
  Complex c;
  c.real = 1.0, c.imag = 0.5;  // 1 + 0.5i

  Complex d;
  d.real = c.real * 2, d.imag = c.imag * 2;  // d = c * 2;
  printf("%f + %fi\n", d.real, d.imag);
  return 0;
}
```

C Structure Example : Complex Number

- Define constructors for the Complex class.

```
struct Complex {
  double real;
  double imag;

  Complex() : real(0.0), imag(0.0) {}
  Complex(const Complex& c) : real(c.real), imag(c.imag) {}
  Complex(double r, double i) : real(r), imag(i) {}
};

int main() {
  // c.real = 1.0, c.imag = 0.5;
  Complex c(1.0, 0.5);  // 1 + 0.5i
  Complex c0 = c, c1(c);

  // d.real = c.real * 2, d.imag = c.imag * 2;
  Complex d(c.real * 2, c.imag * 2);  // d = c * 2;
  printf("%f + %fi\n", d.real, d.imag);
  return 0;
}
```

C Structure Example : Complex Number

- Add some member functions.

```
struct Complex {
  double real;
  double imag;

  Complex() : real(0.0), imag(0.0) {}
  Complex(const Complex& c) : real(c.real), imag(c.imag) {}
  Complex(double r, double i) : real(r), imag(i) {}

  Complex Add(const Complex& c) const {
    return Complex(real + c.real, imag + c.imag);
  }
  Complex Multiply(const Complex& c) const {
    return Complex(real * c.real - imag * c.imag,
                   real * c.imag + imag * c.real);
  }
  void Print() const { printf("%f + %fi", real, imag); };
};

int main() {
  Complex c(1.0, 0.5);  // 1 + 0.5i
  Complex d = c.Multiply(Complex(2, 0));  // d = c * 2;
  d.Print();
  return 0;
}
```

C Structure Example : Complex Number

- Before and after from the user's perspective :

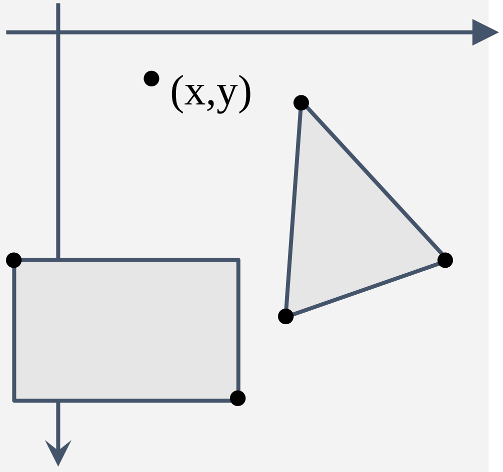
```
int main() {
  Complex c;
  c.real = 1.0, c.imag = 0.5;  // 1 + 0.5i

  Complex d;
  d.real = c.real * 2, d.imag = c.imag * 2;  // d = c * 2;
  printf("%f + %fi\n", d.real, d.imag);
  return 0;
}
```

```
int main() {
  Complex c(1.0, 0.5);  // 1 + 0.5i
  Complex d = c.Multiply(Complex(2, 0));  // d = c * 2;
  d.Print();
  return 0;
}
```

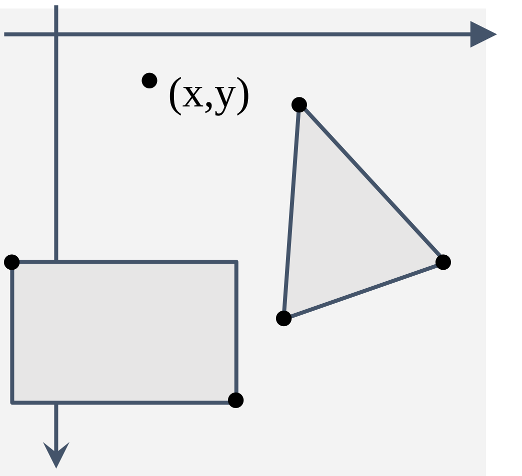
C Structure Example : Shapes

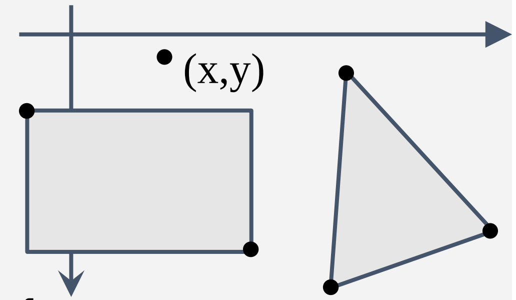
- Structures representing various 2D shapes :

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

struct Line {
  Point p[2];
}

struct Triangle {
  Point p[3];
};

struct Rectangle {
  Point top_left, bottom_right;
};

// Why not struct Rectangle { Point p[4]; }; ?
```



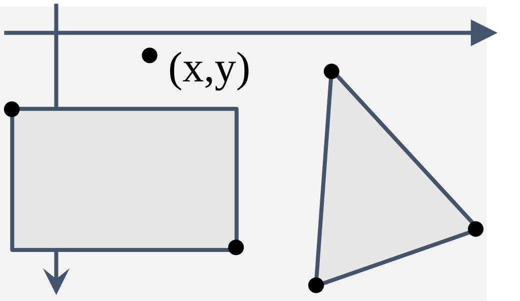
```cpp
struct Point { double x, y; };
struct Line { Point p[2]; };
struct Triangle { Point p[3]; };
struct Rectangle { Point top_left, bottom_right; };

// Compute the length of a line.
double Length(const Line& line);

// Compute the perimeter of a triangle.
double Perimeter(const Triangle& tri);

// Compute the area of a triangle.
double Area(const Triangle& tri);

// Compute the perimeter of a rectangle.
double Perimeter(const Rectangle& rect);

// Compute the area of a rectangle.
double Area(const Rectangle& rect);
```






```
struct Point { double x, y; };
struct Line { Point p[2]; };
struct Triangle { Point p[3]; };
struct Rectangle { Point top_left, bottom_right; };

#include <math.h>

// Distance between two points.
static double Distance(const Point& p0, const Point& p1) {
  const double dx = p1.x - p0.x;
  const double dy = p1.y - p0.y;
  return sqrt(dx * dx + dy * dy);
}

// Compute the length of a line.
double Length(const Line& line){
  return Distance(line.p[0], line.p[1]);
}

// Compute the perimeter of a triangle.
double Perimeter(const Triangle& tri){
  return Distance(tri.p[0], tri.p[1]) + Distance(tri.p[1], tri.p[2]) +
         Distance(tri.p[2], tri.p[0]);
}

// Compute the perimeter of a rectangle.
double Perimeter(const Rectangle& rect){
  return 2 * (fabs(rect.bottom_right.x - rect.top_left.x) +
              fabs(rect.bottom_right.y - rect.top_left.y));
}

// Compute the area of a triangle.
double Area(const Triangle& tri);

// Compute the area of a rectangle.
double Area(const Rectangle& rect);
```

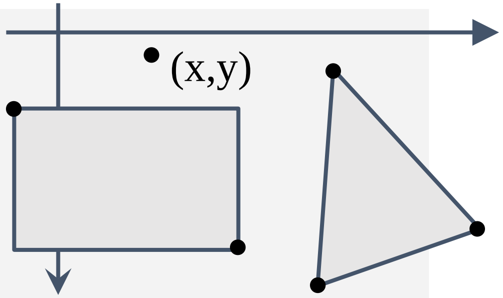
```
...
```




```
// Compute the area of a triangle.
double Area(const Triangle& tri){
  return 0.5 * fabs(
      (tri.p[1].x - tri.p[0].x) * (tri.p[2].y - tri.p[0].y) -
      (tri.p[2].x - tri.p[0].x) * (tri.p[1].y - tri.p[0].y));
}

// Compute the area of a rectangle.
double Area(const Rectangle& rect) {
  return fabs((rect.bottom_right.x - rect.top_left.x) *
              (rect.bottom_right.y - rect.top_left.y));
}
```

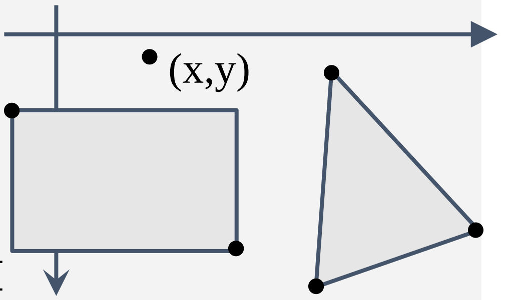
C Structure Example : Shapes

- Use member functions:




```
struct Point {
  double x, y;
  // Distance between two points.
  double Distance(const Point& p) const;
  static double Distance(const Point& p0, const Point& p1);
};

struct Line {
  Point p[2];
  double Length() const;  // Length of the line.
}

struct Triangle {
  Point p[3];
  double Perimeter() const;  // Perimeter of the triangle.
  double Area() const;       // Area of the triangle
};

struct Rectangle {
  Point top_left, bottom_right;
  double Perimeter() const;  // Perimeter of the rectangle.
  double Area() const;       // Area of the rectangle.
};
```

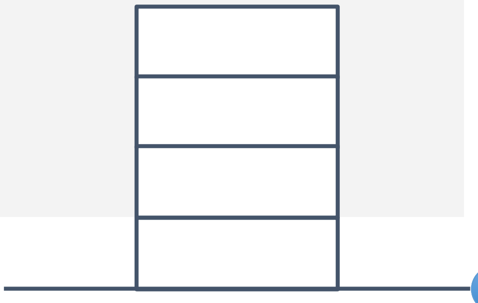
```
#include <math.h>

double Point::Distance(const Point& p) const {
  const double dx = p.x - x, dy = p.y - y;
  return sqrt(dx * dx + dy * dy);
}

double Point::Distance(const Point& p0, const Point& p1) {
  const double dx = p0.x - p1.x, dy = p0.y - p1.y;
  return sqrt(dx * dx + dy * dy);
}

double Line::Length() const {
  return p[0].Distance(p[1]);
}

double Triangle::Perimeter() const {
  return p[0].Distance(p[1]) + p[1].Distance(p[2])
      + p[2].Distance(p[0]);
}

double Triangle::Area() const {
  return 0.5 * fabs((p[1].x - p[0].x) * (p[2].y - p[0].y) -
                    (p[2].x - p[0].x) * (p[1].y - p[0].y));
}

double Rectangle::Perimeter() const {
  return 2 * (fabs(bottom_right.x - top_left.x) +
              fabs(bottom_right.y - top_left.y));
}

double Rectangle::Area() const {
  return fabs((bottom_right.x - top_left.x) *
              (bottom_right.y - top_left.y));
}
```



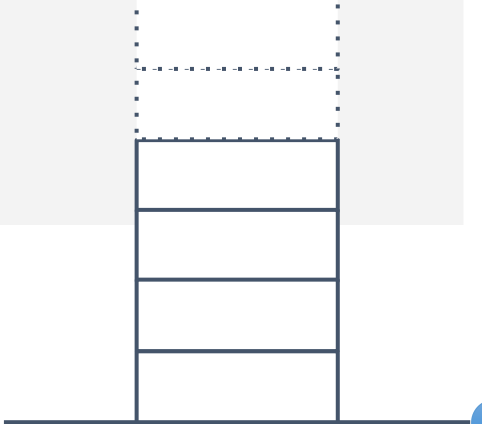
C++ Class Example : Stack

- Stack : Last In First Out (LIFO)

```
class Stack {
 public:
  Stack() : num_data_(0), data_(NULL) {}
  ~Stack() { delete[] data_; }

  void Push(int value);
  void Pop() { if (num_data_ > 0) --num_data_; }
  int Top() const { return data_[num_data_ - 1]; }  // TODO: check NULL.
  bool IsEmpty() const { return num_data_ <= 0; }

 private:
  int num_data_;
  int* data_;
};

void Stack::Push(int value) {
  int* new_data = new int[num_data_ + 1];
  for (int i = 0; i < num_data_; ++i) {
    new_data[i] = data_[i];
  }
  delete[] data_;
  data_ = new_data;
  data_[num_data_] = value;
  ++num_data_;
}
```



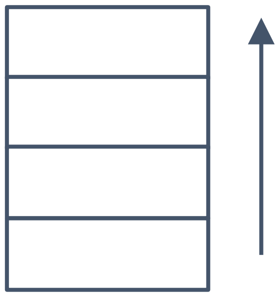
```
class Stack {
 public:
  Stack() : num_data_(0), capacity_(0), data_(NULL) {}
  ~Stack() { delete[] data_; }

  void Push(int value);
  void Pop() { if (num_data_ > 0) --num_data_; }
  int Top() const { return data_[num_data_ - 1]; }
  bool IsEmpty() const { return num_data_ <= 0; }

 private:
  int num_data_, capacity_;
  int* data_;
};

void Stack::Push(int value) {
  if (num_data_ >= capacity_) {
    const int new_capacity = num_data_ + 1;
    int* new_data = new int[new_capacity];
    for (int i = 0; i < num_data_; ++i) {
      new_data[i] = data_[i];
    }
    delete[] data_;
    data_ = new_data;
    capacity_ = new_capacity;
  }
  data_[num_data_] = value;
  ++num_data_;
}
```



C++ Class Example : Queue

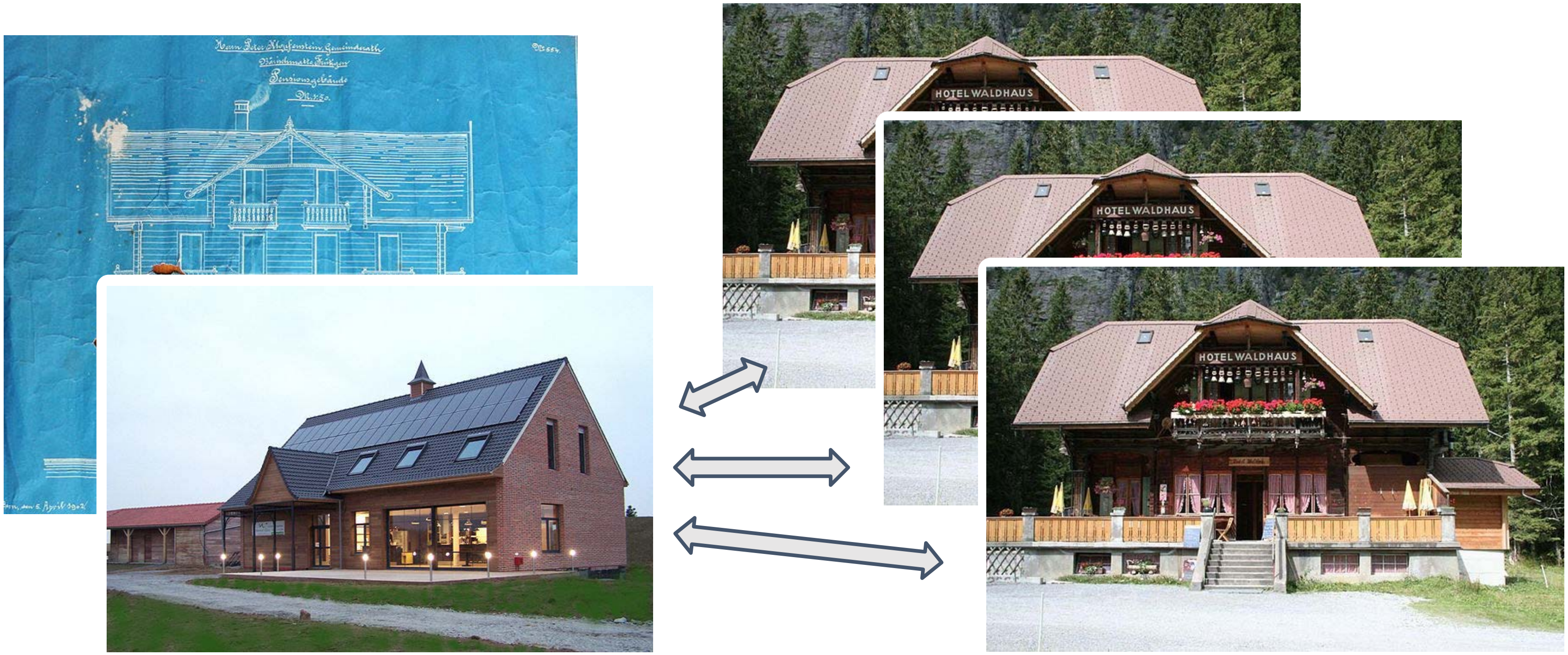
- Queue : First In First Out (FIFO)

```
class Queue {
 public:
  Queue();
  ~Queue();

  void Push(int value);
  void Pop();
  int Front() const;
  int Back() const;
  bool IsEmpty() const;
};
```



Class Static Members

- Static members are about the class, not individual instances.
  - Static member variables are shared by all instances.
  - Static member functions do not have any associated instance, thus only can see the static member variables.



Class Static Members

- Classes can have static member functions and variables.
- Static members are about the class, not the instances.
  - Use the keyword static to specify static members.
  - In static member functions, no this pointer is defined.
  - Static member variables are like global variables, but only for the class.
- To access them, use `ClassName::MemberName`.

Static Member Function Example

```
struct Complex {
  double real;
  double imag;

  Complex() : real(0.0), imag(0.0) {}
  Complex(const Complex& c) : real(c.real), imag(c.imag) {}
  Complex(double r, double i) : real(r), imag(i) {}

  ...

  static Complex Add(const Complex& c1, const Complex& c2) {
    return Complex(c1.real + c2.real, c1.imag + c2.imag);
  }
};

int main() {
  Complex c(1.0, 0.5);  // 1 + 0.5i
  Complex d = Complex::Add(c, Complex(2, 1));  // d = c + (2 + 1i);
  d.Print();
  return 0;
}
```

Static Member Variable Example

```
class CountInstance {
 public:
  CountInstance() { ++count_; PrintCount("construct: "); }
  ~CountInstance() { --count_; PrintCount("destruct:  "); }

  void PrintCount(const string& msg) const { cout << msg << count_ << endl; }
 private:
  static int count_;
};

int CountInstance::count_ = 0;

int main() {
  CountInstance instance;
  for (int i = 0; i < 2; ++i) {
    CountInstance inner_instance;
    // Do nothing.
  }
  return 0;
}
```

```
construct: 1
construct: 2
destruct:  1
construct: 2
destruct:  1
destruct:  0
```

Static Member Example

```
struct MyClass {
  MyClass(double x, double y) : x_(x), y_(y) {}
  void DoSomething();
  static void Prepare();

  double x_, y_;
  static int iter_;
};

int MyClass::iter_ = 0;  // Definition of MyClass::iter_.

void MyClass::DoSomething() {
  for (int i = 0; i < iter_; ++i) cout << x_ + y_ << endl;
}

void MyClass::Prepare() {
  x_ = y_ = 0.0;  // Error!
  iter_ = 10;     // OK.
}

int main() {
  MyClass::Prepare();
  MyClass a;
  a.DoSomething();
  cout << MyClass::iter_ << ", " << a.x_ << endl;
  return 0;
}
```

Static Member Example

```
struct MyClass {
  MyClass(double x, double y) : x_(x), y_(y) {}
  void DoSomething();
  static void Prepare(MyClass* arg);

  double x_, y_;
  static int iter_;
};

int MyClass::iter_ = 0;  // Definition of MyClass::iter_.

void MyClass::DoSomething() {
  for (int i = 0; i < iter_; ++i) cout << x_ + y_ << endl;
}

void MyClass::Prepare(MyClass* arg) {
  arg->x_ = arg->y_ = 0.0;  // OK.
  iter_ = 10;               // OK.
}

int main() {
  MyClass a;
  a.DoSomething();
  MyClass::Prepare(&a);
  cout << MyClass::iter_ << ", " << a.x_ << endl;
  return 0;
}
```

Static Member Example

```
class Singleton {
 public:
  static Singleton* GetInstance();
  // Some useful member functions here..

 private:
  Singleton() { }
  static Singleton* instance_;
};
```

```
Singleton* Singleton::instance_ = NULL;

Singleton* Singleton::GetInstance() {
  if (instance_ == NULL) instance_ = new Singleton;
  return instance_;
}
```

```
int main() {
  Singleton a_instance;  // Error!
  Singleton* ptr = Singleton::GetInstance();
  // Do something.
  return 0;
}
```

Basic Class Design 3

- Hide all data members, unless it is absolutely required (hardly it is).
- Make member functions meaningful and atomic.
- Use `const` as much as possible.
- Use (const) reference or pointers in function parameters, especially when passing a class instance.

- Only simple initialization in constructors.
    - Make a separate setup function for complex initializations, especially when it may fail.
- Use static members only when necessary.
    - Class utility functions that do not need to access data members are often implemented as static functions.

Chapter Summary

- Class vs instance
- Member variables and functions
- Access control : public, private, protected
    - Class vs structure
- Const and reference
- Constructor and destructor
- Static members


Q&A