Classes and Objects

Week 2

Topics

- Abstract Data Types
- Object-Oriented Programming
- Introduction to Classes
- Creating and Using Objects
- Defining Member Functions
- Constructors
- Destructors
- Private Member Functions
- Passing Objects to Functions
- Object Composition
- Separating Class Specification, Implementation, and Client Code
- Structures
- More About Enumerated Data Types

Abstract Data Types and Abstraction

Abstract Data Types are programmer-created data types that specify the legal values that can be stored and the operations that can be done on the values

- The user of an abstract data type (ADT) does not need to know any implementation details (e.g., how the data is stored or how the operations on it are carried out)
- Abstraction allows a programmer to design a solution to a problem and to use data items without concern for how the data items are implemented
  - A. To use the pow function, you need to know what inputs it expects and what kind of results it produces
  - B. You do not need to know how it works

- Abstraction: a definition that captures general characteristics without details
- An abstract triangle is a 3-sided polygon. A specific triangle may be scalene, isosceles, or equilateral
- Data Type: defines the kind of values that can be stored and the operations that can be performed on the values

Object-Oriented Programming

- Procedural programming uses variables to store data, and focuses on the processes/ functions that occur in a program. Data and functions are separate and distinct.
- Object-oriented programming is based on objects that encapsulate the data and the functions that operate with and on the data.
- object: software entity that combines data and functions that act on the data in a single unit
- attributes: the data items of an object, stored in member variables
- member functions (methods): procedures/ functions that act on the attributes of the class

OOP Language

Data Hiding
Encapsulation
Inheritance
Polymorphism

Data Hiding and Encapsulation

- data hiding: Data hiding is a software development technique specifically used in object-oriented programming (OOP) to hide internal object details (data members).
- Data hiding ensures exclusive data access to class members and protects object integrity by preventing unintended or intended changes.
- Data hiding also reduces system complexity for increased robustness by limiting interdependencies between software components.
- encapsulation: the bundling of an object's data and procedures into a single entity
- Data hiding only hides class data components, whereas data encapsulation hides class data parts and private methods.
- Protection – Member functions provide a layer of protection against inadvertent or deliberate data corruption
- Need-to-know – A programmer can use the data via the provided member functions. As long as the member functions return correct information, the programmer needn't worry about implementation details.

Classes

- Class: a programmer-defined data type used to define objects
- It is a pattern or a template for creating objects
- Used to control access to members of the class.
- Each member is declared to be either
  - public: can be accessed by functions outside of the class
  - private: can only be called by or accessed by functions that are members of the class
- Access specifiers:
- Can be listed in any order in a class
- Can appear multiple times in a class
- If not specified, the default is private

Class declaration format:

```
class className
{
    declaration;
    declaration;
};
```

```
class Square {
private:
   int side;

public:
   void setSide(int s) {
         side = s;
    }

    int getSide(){
        return side;
    }
};
```

Access Specifier

Creating and Using Objects

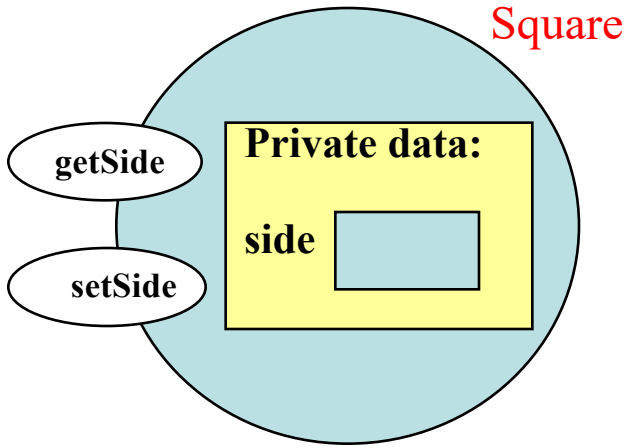
- An object is an instance of a class
- It is defined just like other variables

```
Square sq1, sq2;
```

- It can access members using dot operator

```
sq1.setSide(5);
cout << sq1.getSide();
```

- Acessor, get, getter function: uses but does not modify a member variable (`getSide`)
- Mutator, set, setter function: modifies a member variable (`setSide`)



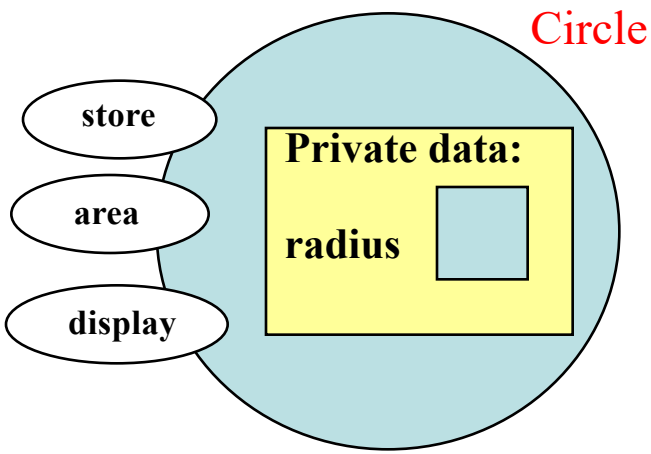

```
#include <iostream>
using namespace std;

class Square {
private:
    int side;

public:
    void setSide(int s) {
            side = s;
    }

    int getSide() {
        return side;
    }
};

int main ( ) {
    Square sq1, sq2;
    sq1.setSide(5);
    cout << sq1.getSide()
            << endl;
    return 0;
}
```

Creating and Using Objects

```
class circle
{
   private:
      double radius;

   public:
      void store(double);
      double area(void);
      void display(void);
};
```

```
// member function definitions
void circle::store(double r) {
    radius = r;
}
double circle::area(void) {
    return 3.14*radius*radius;
}
void circle::display(void) {
    cout << “r = “ << radius << endl;
}
```

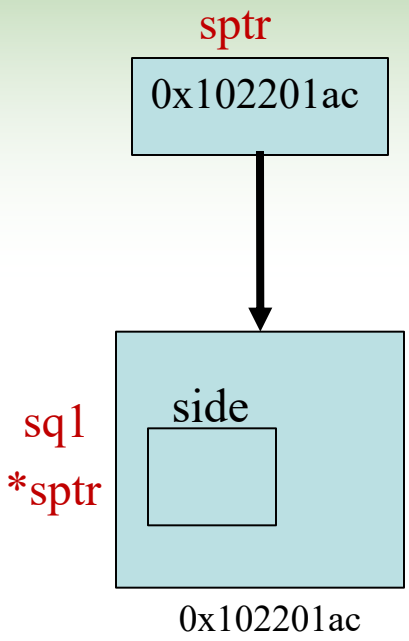
```
int main( ) {
    circle c;
    c.store(5.0);
    cout << "The area of circle c is ”
            << c.area()
            << endl;
    c.display();
}
```



Creating and Using Objects: **Pointers**

```
#include <iostream>
using namespace std;

class Square
{
private:
    int side;
public:
    void setSide(int s) {
        side = s;
    }
    int getSide() {
        return side;
    }
};
```

```
int main ( )
{
    Square sq1;
    Square *sptr;
    sptr = &sq1;

    sptr->setSide(5);
    cout << sptr->getSide()
         << endl;
    return 0;
}
```




```
sptr->setSide(5);
cout <<  sptr->getSide() << endl;
```

Pointer format ->

Using the .

```
(*sptr).setSide(5);
cout <<  (*sptr).getSide() << endl;
```

Creating and Using Objects: References

```
#include <iostream>
using namespace std;

class Square
{
private:
    int side;
public:
    void setSide(int s) {
        side = s;
    }
    int getSide() {
        return side;
    }
};
```

```
int main ( )
{
    Square sq1;
    Square &ref = sq1;

    ref.setSide(5);
    cout << ref.getSide()
         << endl;
    return 0;
}
```

Reference is initialized when declared

sq1 or ref can be used to refer to the same object

Defining Member Functions: Behavior

Member functions are part of a class declaration

- You can place entire function definition inside the class declaration
- Member functions defined inside the class declaration are called inline functions
- Only very short functions, like setSide and getSide, should be inline functions

- You can place just the prototype inside the class declaration and write the function definition after the class
- In the function definition, precede the function name with the class name and scope resolution operator (::)

```
class Square {
private:
    int side;
public:
    void setSide(int);
    int getSide( );
};

void Square::setSide(int s) {
    side = s;
}
int Square::getSide(){
    return side;
}
```

```
class Square {
private:
    int side;

public:
    void setSide(int s) {
            side = s;
    }

    int getSide() {
        return side;
    }
};
```

Tradeoffs of Inline vs. Regular Member Functions

- When a regular function is called, control passes to the called function
- the compiler stores return address of call, allocates memory for local variables, etc.
- Code for an inline function is copied into the program in place of the call when the program is compiled
- This makes a larger executable program, but there is less function call overhead, and possibly faster execution

Suggestion: If possible, use member variables to calculate a value to be returned, as opposed to storing the calculated value. This minimizes the likelihood of stale data.

Stale Data: an object's members are filled with information from a database, but the underlying **data** in the database has changed since the object was filled.

- Conventions:
- Member variables are usually **private**
- Accessor and mutator functions are usually **public**
- Use '**get**' in the name of accessor functions, '**set**' in the name of mutator functions

Object Initialization

- A constructor is a member function is automatically called when an object of the class is created
- It can be used to initialize data members
- In most cases, it must be a public member function
- It must be named the same as the class
- It must have no return type
- A class can have more than 1 constructor
- Three types of Constructors:
  - **Default Constructor**
  - **Constructor with parameters**
  - **Copy Constructor**
- Overloaded constructors in a class must have different parameter lists

  Square();

  Square(int);

```
class Square
{
private:
    int side;
public:
    Square ( ) {
        side = 0;
    }

    Square (int v) {
        side = v;
    }

   void setSide(int v) {
    side = v;
    }
    int getSide(){
         return side;
    }
};
```

```
class Square
{
private:
    int side;
public:
    Square ( );
    Square (int v);
    void setSide(int v);
    int getSide();
};

Square::Square ( ) {
      side = 0;
}
Square::Square (int v) {
      side = v;
}

Void Square:: setSide(int v){
     side = v;
}
int Square::getSide(){
     return side;
}
```


The Default Constructor

- Constructors can have any number of parameters, including none
- A default constructor is one that takes no arguments either due to
- No parameters or
- All parameters have default values
- When a class is declared with no constructors, the compiler automatically assumes default constructor and copy constructor for it.
- If a class has any programmer-defined constructors, it must have a programmer - defined default constructor
- To create an object using the default constructor, use no argument list and no ()

  Square square1;

- To create an object using a constructor that has parameters, include an argument list

  Square square2(8);

```
class Square
{
private:
    int side;

public:
    Square ( ) {
        side = 0;
    }

    Square (int v) {
        side = v;
    }

    void setSide(int v) {
     side = v;
     }
     int getSide(){
          return side;
     }
};
```

=

```
class Square
{
private:
    int side;

public:
    Square ( int s = 0) {
        side = s;
    }

    void setSide(int v) {
     side = v;
     }
     int getSide(){
          return side;
     }
};
```

The Default Constructor

```
class Rectangle
{
 private:
    int width;
    int length;
public:
    Rectangle(int w, int l) {
        width =w; length=l;
    }
    void set(int w, int l);
    int area();
}
```

If any constructor with any number of parameters is declared, no default constructor will exist, unless you define it.

```
Rectangle r4;          // error
```

Initialize with constructor

```
Rectangle r5(60,80);
Rectangle *r6 = new Rectangle(60,80);
```

The Default Constructor: Pointers and Dynamic Allocation

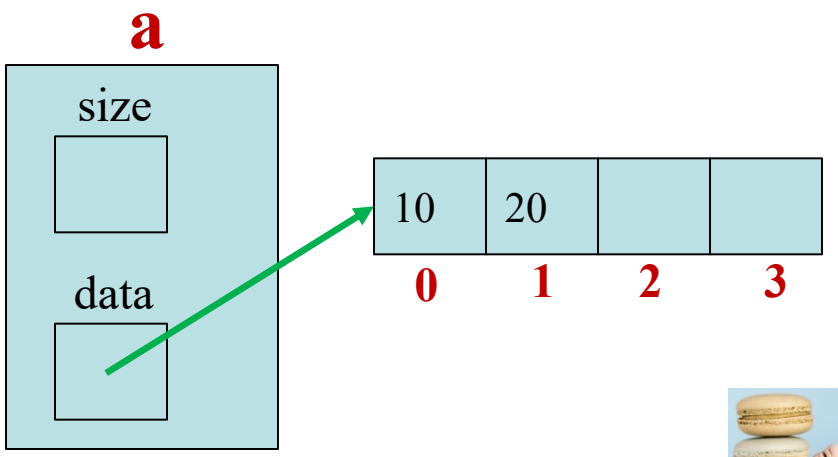
```
#include <iostream>
using namespace std;

class myArray
{
private:
    int size;
    int *data;
public:
    myArray(){
        size = 4;
        data = new int[size];
    }

    void setItem(int index, int value) {
        data[index] = value;
    }
    int getItem(int index) {
        return data[index];
    }
};
```

```
int main ( )
{
    myArray a;
    a.setItem(0, 10);
    a.setItem(1, 20);
    cout << a.getItem(1)
         << endl;
    return 0;
}
```

```
20
```

Define a dynamic data (pointer to an int)

Dynamically allocate memory for data



Destructors : Cleanup of an Object

- Are public member functions that are automatically called when objects are destroyed
- The destructor name is ~className
- It has no return type
- It takes no arguments
- Only 1 destructor is allowed per class (i.e., it cannot be overloaded)
- It is executed automatically when the object goes out of scope

```
#include <iostream>
using namespace std;

class Square {
private:
    int side;
public:
    Square (int s = 1) {
        cout << "Const ..."
             << endl;
        side = s;
    }
    ~Square(){
        cout << "Destr ..."
             << endl;
    }
    void setSide(int v){
        side = v;
    }
    int getSide(){
        return side;
    }
};
```

```
int main()
{
    Square sq(6);
    cout << sq.getSide()
         << endl;
    return 0;
}
```

```
Const ...
6
Destr ...
```

Destructors : Pointer Data

```
class myArray
{
private:
    int size;
    int *data;
public:
    myArray(){
        size = 4;
        data = new int[size];
    }
    ~myArray ( ) {
        delete data;
    }
    void setItem(int index, int value) {
        data[index] = value;
    }
    int getItem(int index) {
        return data[index];
    }
};
```

```
int main ( )
{
    myArray a;
    a.setItem(0, 10);
    a.setItem(1, 20);
    cout << a.getItem(1)
         << endl;
    return 0;
}
```

Allocate memory dynamically

Must free the allocated memory to the heap

Private Member Functions

- A private member function can only be called by another member function of the same class
- It is used for internal processing by the class, not for use outside of the class

```
int main()
{
    Square sq(1);
    sq.setSide(5);
    sq.incr();      // will cause compile error
    cout << sq.getSide()
         << endl;
    return 0;
}
```

```
||=== Build: Debug in week2 (compiler: GNU GCC Compiler) ===|
D:\TCP1201_1820\week2\main.cpp||In function 'int main()':|
D:\TCP1201_1820\week2\main.cpp|7|error: 'void Square::incr()' is private|
D:\TCP1201_1820\week2\main.cpp|32|error: within this context|
||=== Build failed: 2 error(s), 0 warning(s) (0 minute(s), 0 second(s)) ===|
```

```
Const ...
6
Destr ...
```

```
class Square {
private:
    int side;

    void incr ( ) {
        side++;
    }

public:
    Square (int s = 1) {
        cout << "Const ...“ << endl;
        side = s;
    }
    ~Square(){
        cout << "Destr ...“ << endl;
    }
    void setSide(int v){
        side = v;
        incr();                    // ok
    }
    int getSide(){
        return side;
    }
};
```

Passing Objects to Functions by Value

- A class object can be passed as an argument to a function.
- When it is passed by value, the function makes a local copy of the object. The original object in calling environment is unaffected by actions in the function.
- Using a value parameter for an object can slow down a program and waste space
- When passed by reference, the function can use 'set' functions to modify the object in the calling environment.
- Using a reference parameter speeds up the program. It allows the function to modify data in the parameter, which may not be desirable.

```
class Square {
private:
    int side;
public:
    Square (int s = 1) {
        cout << "Const ..." << endl;
        side = s;
    }
    ~Square(){
        cout << "Destr ..." << endl;
    }
    void setSide(int v) { side = v; }
    int getSide() { return side; }
};
void print (Square s){
    cout << s.getSide()
         << endl;
    s.setSide(20);
}
int main() {
    Square sq(10);
    print (sq);
    cout << sq.getSide() << endl;
    return 0;
}
```

```
Const ...
10
Destr ...
10
Destr ...
```

Passing Objects to Functions by Reference

- A class object can be passed as an argument to a function.
- When passed by reference, the function can use ‘set’ functions to modify the object in the calling environment.
- Using a reference parameter speeds up the program. It allows the function to modify data in the parameter, which may not be desirable.

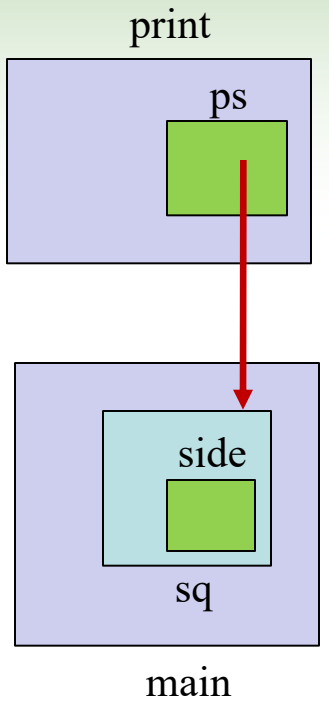
```
class Square {
private:
    int side;
public:
    Square (int s = 1) {
        cout << "Const ..." << endl;
        side = s;
    }
    ~Square(){
        cout << "Destr ..." << endl;
    }
    void setSide(int v) { side = v; }
    int getSide() { return side; }
};
void print (Square &s) {
    cout << s.getSide()
         << endl;
    s.setSide(20);
}
int main() {
    Square sq(10);
    print (sq);
    cout << sq.getSide() << endl;
    return 0;
}
```

```
Const ...
10
20
Destr ...
```

Passing Objects to Functions by Pointer

```
#include <iostream>
using namespace std;

class Square {
private:
    int side;
public:
    Square (int s = 1) {
        cout << "Const ..." << endl;
        side = s;
    }
    ~Square(){
        cout << "Destr ..." << endl;
    }
    void setSide(int v){
      side = v;
    }
    int getSide() { return side; }
};
```

```
void print (Square *ps) {

    cout << ps->getSide()
         << endl;

    ps->setSide(20);
}

int main()
{
    Square sq(10);

    print ( &sq );

    cout << sq.getSide() << endl;

    return 0;
}
```




```
Const ...
10
20
Destr ...
```

Notes on Passing Objects

- To save space and time while protecting parameter data that should not be changed, use a const reference parameter

void showData (const Square &s) // header

```
void showData (const Square &s)
{
   cout << s.getSide() << endl;
   // s.setSide(20);   // causes error
}

int main() {
    Square sq(10);
    showData(sq);
    cout << sq.getSide()
         << endl;
    return 0;
}
```

```
Const ...
10
10
Destr ...
```

```
class Square {
private:
    int side;
public:
    Square (int s=1) {
        cout<<"Const ..."<<endl;
        side = s;
    }
    ~Square () {
      cout <<"Destr ..."<<endl;
    }
    void setSide (int v) {
      side = v;
    }
    int getSide () {
      return side;
    }
};
```

Notes on Passing Objects

- In order to for the showData function to call Square member functions, those functions must use const in their prototype and header:

int Square::getSide () const;

```
void showData (const Square &s)
{
   cout << s.getSide() << endl;
}

int main()
{
    Square sq(10);
    showData(sq);
    cout << sq.getSide()
         << endl;
    return 0;
}
```

```
Const ...
10
10
Destr ...
```

```
class Square {
private:
    int side;
public:
    Square (int s=1) {
        cout<<"Const ..."<<endl;
        side = s;
    }
    ~Square () {
      cout <<"Destr ..."<<endl;
    }
    void setSide (int v) {
      side = v;
    }
    int getSide () const {
      // size = 20;    // causes error
      return side;
    }
};
```

Returning an Object from a Function

- A function can return an object

```
Square initSquare();        // prototype
Square s1 = initSquare(); // call
```

- The function must create an object
- for internal use
- to use with return statement

```
class Square {
private:
    int side;
public:
    Square (int s=1) {
        cout<<"Const ..."<<endl;
        side = s;
    }
    ~Square () {
      cout <<"Destr ..."<<endl;
    }
    void setSide (int v) {
      side = v;
    }
    int getSide () const {
       return side;
    }
};
```

```
Square initSquare()
 {
   Square s;
   int inSize;
   cout << "Enter side: ";
   cin >> inSize;
   s.setSide(inSize);
   return s;
 }

int main()
{
    Square sq = initSquare();
    cout << sq.getSide()
         << endl;
    return 0;
}
```

```
Const ...
Enter side: 88
88
Destr ...
```

Separating Class Specifications, Implementations, and Client Code

- Separating the class declaration, member function definitions, and the program that uses the class into separate files is considered good design.
- Place the class declaration in a header file that serves as the class specification file. Name the file classname.h (Square.h)
- Place the member function definitions in a class implementation file. Name the file classname.cpp (for example, Square.cpp)This file should #include the class specification file.
- A client program (client code) that uses the class must #include the class specification file and be compiled and linked with the class implementation file.

Square.h

```
#ifndef SQUARE_H
#define SQUARE_H
class Square {
private:
   int side;
public:
   Square(int s=1);
   ~Square();
   void setSide(int v);
   int getSide()const;
};
#endif
```

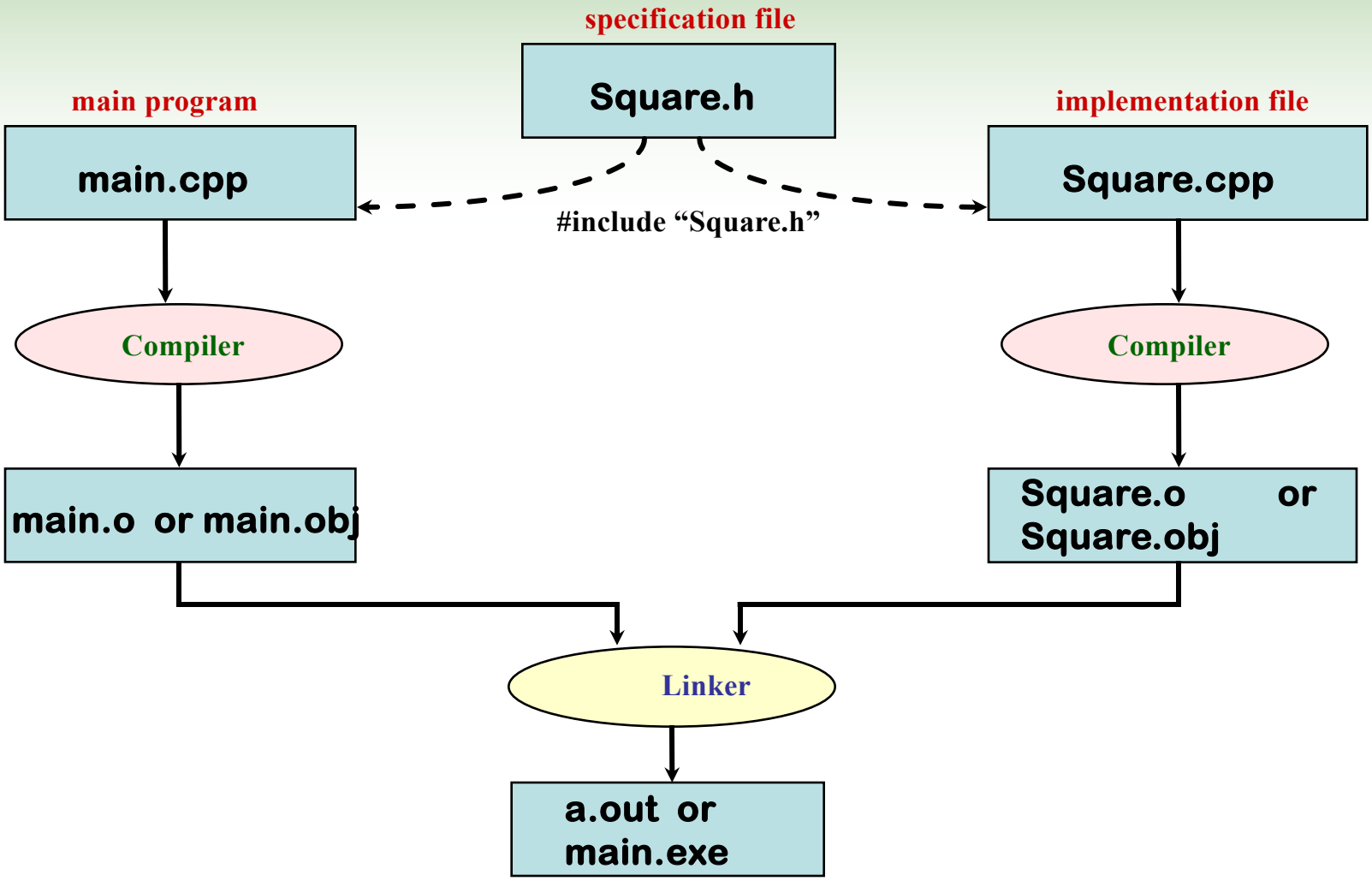
Square.cpp

```
#include <iostream>
#include "Square.h"
using namespace std;

Square::Square(int s){
   cout<<"Const ..."<<endl;
   side = s;
}
Square::~Square(){
   cout <<"Destr ..."<<endl;
}
void Square::setSide(int v){
   side = v;
}
int Square::getSide()const{
    return side;
}
```

main.cpp

```
#include <iostream>
#include "Square.h"
using namespace std;

Square initSquare()
 {
   Square s;
   int inSize;
   cout << "Enter side: ";
   cin >> inSize;
   s.setSide(inSize);
   return s;
 }

int main(){
    Square sq = initSquare();
    cout << sq.getSide()
         << endl;
    return 0;
}
```

Separating Class Specifications, Implementations, and Client Code



Structures

- Structure: A programmer-defined data type that allows multiple variables to be grouped together
- Structure declaration format:

```
struct structure name
{
    type1 field1;
    type2 field2;
        …
    typen fieldn;
};
```

```
struct Student
{
    int studentID;
    string name;
    short year;
    double gpa;
};
```

structure name

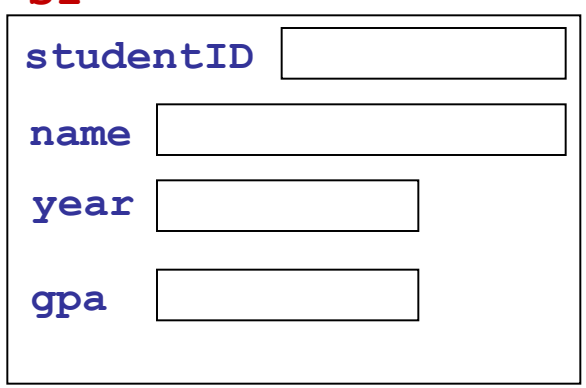
structure members

Note the required ;

- struct names commonly begin with an uppercase letter
- Multiple fields of same type can be declared in a comma-separated list

  string name, address;

- Fields in a structure are all public by default
- A **struct** declaration does not allocate memory or create variables
- To define variables, use structure tag as the type name (**Student s1;**)



Accessing Structure Members

- Use the dot (.) operator to refer to the data members of struct variables
- The member variables can be used in any manner appropriate for their data type
- To display the contents of a struct variable, you must display each field separately, using the dot operator
- Wrong:

```
cout << s1;      // won't work!
```

- Similar to displaying a struct, you cannot compare two struct variables directly:

```
if (s1 >= s2)      // won't work!
```

- Instead, compare member variables:

```
if (s1.gpa >= s2.gpa)      // better
```

```
sharaf
Sami
1234
2010
1234
sharaf
Sami
2010
3.75
```

```
#include <iostream>
using namespace std;

struct Student {
    int studentID;
    string name;
    short year;
    double gpa;
};
void show (Student &st){
   cout << st.studentID << endl
        << st.name << endl
        << st.year << endl
        << st.gpa << endl;
}
int main ( ) {
   Student s;
   getline(cin, s.name);
   cin >> s.studentID;
   cin >> s.year;
   s.gpa = 3.75;
   show(s);
   return 0;
}
```

Initializing Structures

- Structure members cannot be initialized in the structure declaration, because no memory has been allocated yet
- Structure members are initialized at the time a structure variable is created
- You can initialize a structure variable's members with either
- an initialization list
- a constructor
- An initialization list is an ordered set of values, separated by commas and contained in { }, that provides initial values for a set of data members

```
{12, 6, 3}    // initialization list
              // with 3 values
```

```
struct Student                  // Illegal
 {                              // initialization
   int studentID = 1145;
   string name = "Alex";
   short year = 1;
   float gpa = 2.95;
 };
```

- The order of list elements matters: The first value initializes first data member, second value initializes second data member, etc.
- The elements of an initialization list can be constants, variables, or expressions

```
{12, W, L/W + 1}
// initialization list
// with 3 items
```

- You can initialize just some members, but you cannot skip over members

```
Dimensions box1 = {12,6};   //OK
Dimensions box2 = {12,,3}; //illegal
```

Initializing Structures

```
#include <iostream>
using namespace std;

struct Student
{
    int studentID;
    string name;
    short year;
    double gpa;
};
void show (Student &st){
   cout << st.studentID << endl
        << st.name << endl
        << st.year << endl
        << st.gpa << endl;
}
```

```
int main()
{
    string name = "sharaf";
    Student s1 = {10, name, 2010, 3.5};
    Student s2{11,"Sami",2011,3.5};
    Student s3{12,"Horani"};
    show(s1);
    show(s2);
    show(s3);
    return 0;
}
```

```
10
Sharaf
2010
3.5
11
Sami
2011
3.5
12
Horani
0
0
```

- You can't omit a value for a data member without omitting values for all following members
- It does not work on most modern compilers if the structure contains objects, e.g., like string objects

Using a Constructor to Initialize Structure Members

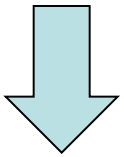
- Similar to a constructor for a class
- the name is the same as the name of the struct
- it has no return type
- it is used to initialize data members
- It is normally written inside the struct declaration

```
#include <iostream>
#include "Square.h"
using namespace std;

struct Student
{
    int studentID;
    string name;
    short year;
    double gpa;

    Student ( ) {
        name = "Dfault";
        year = 1990;
        gpa = 2.0;
        studentID = 0;
    }
};
```

```
void show (Student &st){
    cout << st.studentID << endl
         << st.name << endl
         << st.year << endl
         << st.gpa << endl;
}

int main()
{
    Student s3;
    show(s3);
    return 0;
}
```

```
0
Dfault
1990
2
```

```
int main()
{
    Student s3 (1,"abc",2,3.1);
    show(s3);
    return 0;
}
```



```
Student (int id, string nm, short y, double gp) {
        name = nm;
        year = y;
        gpa = gp;
        studentID = id;
}
```

```
1
abc
2
3.1
```

Nested Structures

- A structure can have another structure as a member.

```
struct PersonInfo
{  string name,
          address,
          city;
};
struct Student
{  int         studentID;
   PersonInfo  pData;
   short       year;
   double      gpa;
};
```

```
void show (Student &st) {
    cout << st.studentID << endl
         << st.pData.name << endl
         << st.pData.address << endl
         << st.pData.city << endl
         << st.year << endl
         << st.gpa << endl;
}
int main()  {
    Student s3 {10, {"sharaf","MMU","cyberjaya"} ,2010,3.4};
    show(s3);
    return 0;
}
```

```
10
sharaf
MMU
cyberjaya
2010
3.4
```

```
void printInfo(PersonInfo &p) {
    cout << p.name << endl
         << p.address << endl
         << p.city << endl;
}
```

```
int main()
{
    Student s3
{10,{"sharaf","MMU","cyberjaya"},2010,3.4};
    show(s3);
    printInfo(s3.pData);
    return 0;
}
```

```
10
sharaf
MMU
cyberjaya
2010
3.4
sharaf
MMU
cyberjaya
```

More About Enumerated Data Types

- Additional ways that enumerated data types can be used:
- Data type declaration and variable definition in a single statement:

```
enum Tree {ASH, ELM, OAK} tree1, tree2;
```

- Assign an int value to an enum variable:

```
enum Tree {ASH, ELM, OAK} tree1;
tree1 = static_cast<Tree>(1);  // ELM
```

- Assign the value of an enum variable to an int:

```
enum Tree {ASH, ELM, OAK} tree1;
tree1 = ELM;
int thisTree = tree1; // assigns 1
int thatTree = OAK;   // assigns 2
```

- Assign the result of a computation to an enum variable

```
#include <iostream>
using namespace std;

int main()
{
   enum Tree {ASH, ELM, OAK} tree1;

   tree1 = ELM;

   int thisTree = tree1;  // assigns 1

   int thatTree = OAK;    // assigns 2

  Tree tree2 = static_cast<Tree> (tree1 + 1);

   cout <<  tree1  << endl;
    return 0;
}
```

1

More About Enumerated Data Types

```
#include <iostream>
using namespace std;

int main()
{
   enum Tree {ASH, ELM, OAK} tree1;
   cout << "Enter tree type \n(0:ASH, 1:ELM, 2:OAK):";
   int tr;
   cin >> tr;
   tree1 = static_cast<Tree>(tr);
   switch(tree1)
   {
    case ASH: cout << "Ash";
              break;
    case ELM: cout << "Elm";
              break;
    case OAK: cout << "Oak";
              break;
   }
    return 0;
}
```

```
Enter tree type
(0:ASH, 1:ELM, 2:OAK):2
Oak
```

```
#include <iostream>
using namespace std;

int main()
{
   enum Day { Monday, Tuesday,
Wednesday, Thursday, Friday,
Saturday, Sunday};

for (Day d1 = Wednesday;
      d1 <= Saturday;
      d1=static_cast<Day>(d1+1))  {

        cout << d1 << ":" ;
    }

    cout << endl;
    return 0;
}
```

```
2:3:4:5:
```