CSE 1062 **Fundamentals of Programming**

Lecture #13

Spring 2016

Computer Science & Engineering Program
The School of EE & Computing
Adama Science & Technology University

Outline

ASTU

Structure

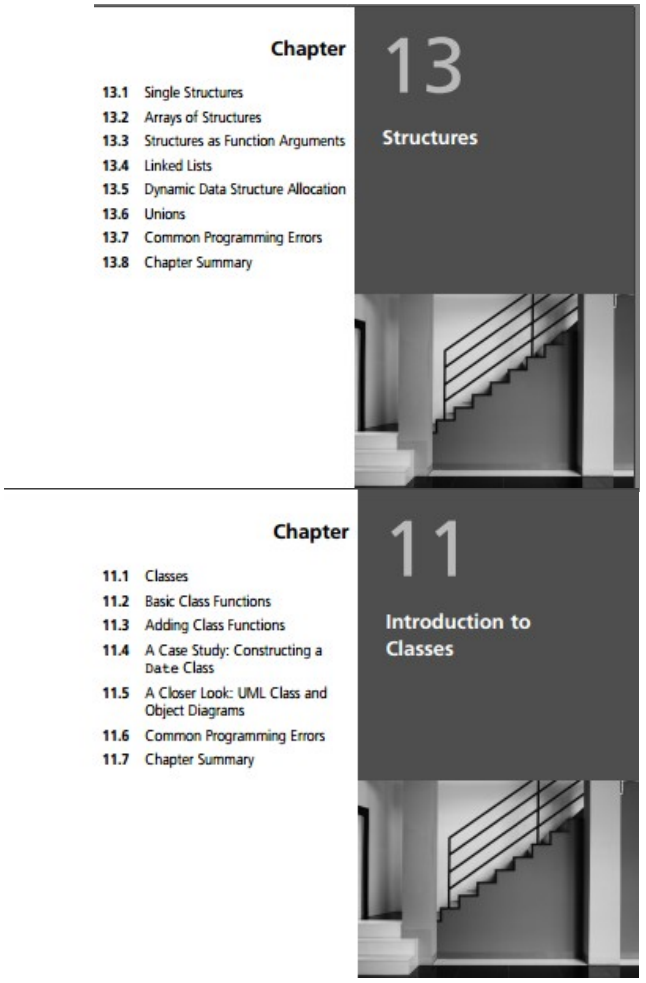
- Single structures
- Arrays of structures
- Structures as function arguments
- Linked lists

Object Oriented Programming I

- Classes
- Basic class functions
- Adding class functions

Case Study: Constructing Date Object

- Reading Assignments
  - Chapter 11 of the text book
  - Chapter 13 of the text book
  - Dynamic Structure Allocation
  - Complex Numbers



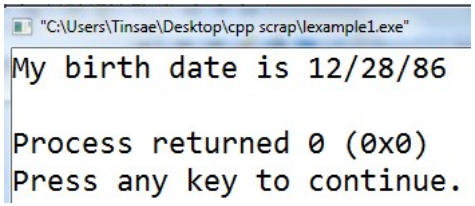
Single Structures

- Creating and using a structure involves two st eps
  - Declare record structure
  - Assign specific values to structure elements
- Declaring a structure requires listing data typ es, data names, and arrangement of data ite ms
- Data items, or fields, are called **members** of a structure
- Assigning data values to members is called **p opulating the structure**

Example 1: Define and Populate a Structure

ASTU

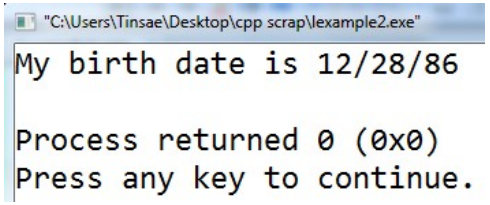
```
1  // A program that defines and populates a structure
2  #include <iostream>
3  using namespace std;
4  int main()
5  {
6      struct
7      {
8          int month;
9          int day;
10         int year;
11     } birth;
12     birth.month = 12;
13     birth.day = 28;
14     birth.year = 86;
15     cout << "My birth date is "
16          << birth.month << '/'
17          << birth.day << '/'
18          << birth.year << endl;
19     return 0;
20 }
```



Example 2: Global Declaration

ASTU

```
1  #include <iostream>
2  using namespace std;
3  struct Date // this is a global declaration
4  {
5      int month;
6      int day;
7      int year;
8  };
9  int main()
10 {
11     Date birth;
12     birth.month = 12;
13     birth.day = 28;
14     birth.year = 86;
15     cout << "My birth date is " << birth.month << '/'
16          << birth.day << '/'
17          << birth.year << endl;
18     return 0;
19 }
```



Arrays of Structures

ASTU

- The real power of structures is realized when the same structure is used for lists of data

ID	Name	Groups
101	Anteneh Alemu	[G1-G2]
201	Desalegn Abebaw	[G3-G6]
303	Tinsae Gizachew	[G9-G10]
405	Kifle Derese	[G13-G18]
590	Desta Zerihun	[G19-G24]
699	Nemera Jira	[G25-G30]
715	Natnael Alemayehu	[G31-G36]
888	Getinet Yilma	[G39-G42]
987	Fanos Jemal	[G43-G46]
1001	Endris Mohammed	[G7-G8], G49-G50]
1002	Prof. Yun Koo Chung	[G51-G52]
1055	Bushra Ali	[11-12],[37-38],[G47-G48]

Arrays of Structures

ASTU

	ID	Name	Groups
1st Structure →	101	Anteneh Alemu	[G1-G2]
2nd Structure →	201	Desalegn Abebaw	[G3-G6]
3rd Structure →	303	Tinsae Gizachew	[G9-G10]
4th Structure →	405	Kifle Derese	[G13-G18]
5th Structure →	590	Desta Zerihun	[G19-G24]
6th Structure →	699	Nemera Jira	[G25-G30]
7th Structure →	715	Natnael Alemayehu	[G31-G36]
8th Structure →	888	Getinet Yilma	[G39-G42]
9th Structure →	987	Fanos Jemal	[G43-G46]
10th Structure →	1001	Endris Mohammed	[G7-G8], [G49-G50]
11th Structure →	1002	Prof. Yun Koo Chung	[G51-G52]
12th Structure →	1055	Bushra Ali	[11-12],[37-38],[G47-G48]

Example 3: Array of Structures

ASTU

```
1   #include <iostream>
2   #include <iomanip>
3   #include <string>
4   using namespace std;
5   const int NUMRECS = 5; // maximum number of records
6   struct CourseRec // this is a global declaration
7   {
8       int id;
9       string name;
10      string groups;
11  };
12  int main()
13  {
14      int i;
15      CourseRec instructor[NUMRECS] =
16      {
17          { 101, "Anteneh, A.", "[G1-G2]"},
18          { 201, "Desalegn, A.", "[G3-G6]"},
19          { 303, "Tinsae, G.", "[G9-G10]"},
20          { 405, "Kifle, D.", "[G13-G18]"},
21          { 590, "Desta, Z.", "[G19-G24]"}
22      };
23      cout << endl; // start on a new line
24      cout << setiosflags(ios::left);  // left-justify the output
25      for (i = 0; i < NUMRECS; i++)
26          cout << setw(5) << instructor[i].id
27               << setw(15) << instructor[i].name
28               << setw(6) << instructor[i].groups << endl;
29      return 0;
```

```
"C:\Users\Tinsae\Desktop\cpp scrap\lexample3.exe"

101  Anteneh, A.    [G1-G2]
201  Desalegn, A.   [G3-G6]
303  Tinsae, G.     [G9-G10]
405  Kifle, D.      [G13-G18]
590  Desta, Z.      [G19-G24]
```

Structures as Function Arguments

ASTU

- Structure members can be passed to a function just like any scalar variable
- Given the structure `emp` definition:

```
struct
{
   int idNum;
   double payRate;
   double hours;
} emp;
```

- Pass a copy of `emp.idNum` to `display()` function:

```
display(emp.idNum);
```

Structures as Function Arguments

- Copies of all structure members can be passed to a function by including the name of the structure as an argument to a called function
  - Example: `calcNet(emp)`

Example 4: Passing by Value

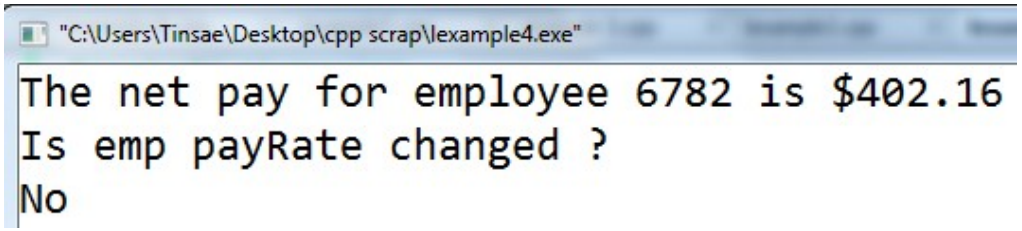
ASTU

```
1  #include <iostream>
2  #include <iomanip>
3  using namespace std;
4  struct Employee // declare a global data type
5  {
6      int idNum;
7      double payRate;
8      double hours;
9  };
10 double calcNet(Employee); // function prototype
11 int main()
12 {
13     Employee emp = {6782, 8.93, 40.5};
14     double netPay;
15     netPay = calcNet(emp); // pass copies of the values in emp
16 // Set output formats
17     cout << setw(10)
18          << setiosflags(ios::fixed)
19          << setiosflags(ios::showpoint)
20          << setprecision(2);
21     cout << "The net pay for employee " << emp.idNum
22          << " is $" << netPay << endl;
23     cout<<"Is emp payRate changed ?\n";
```

Example 4: Passing by Value

ASTU

```
24      if(emp.payRate==8.93)   //still as initialized ?
25          cout<<"No";
26      else
27          cout<<"Yes";
28      return 0;
29  }
30  double calcNet(Employee temp) // temp is of data type Employee
31  {
32      temp.payRate++; //salary increment by the government!!
33      return temp.payRate * temp.hours;
34  }
35
```



Example 5: Passing by Reference

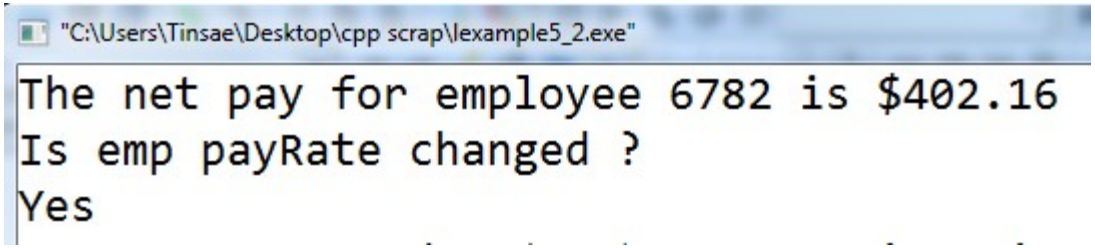
ASTU

- Change line 10

```
10    double calcNet(Employee&); // function prototype
```

- Change line 30 (header of fun. definitio n)

```
30    double calcNet(Employee& temp)
```



Passing a Pointer

- Instead of passing references, pointers can be used
- Function call must take the address of the structure
  - Example: `calcNet(&emp);`
- Function declaration must indicate a pointer argument
  - Example: `calcNet(Employee *pt)`
- Inside function, `pt` argument is used to reference members directly
  - Example: `(*pt).idNum`

Example 6: Passing Pointer

ASTU

- Change line 10

```
10  double calcNet(Employee *); // function prototype
```

- Change line 15

```
15      netPay = calcNet(&emp);
```

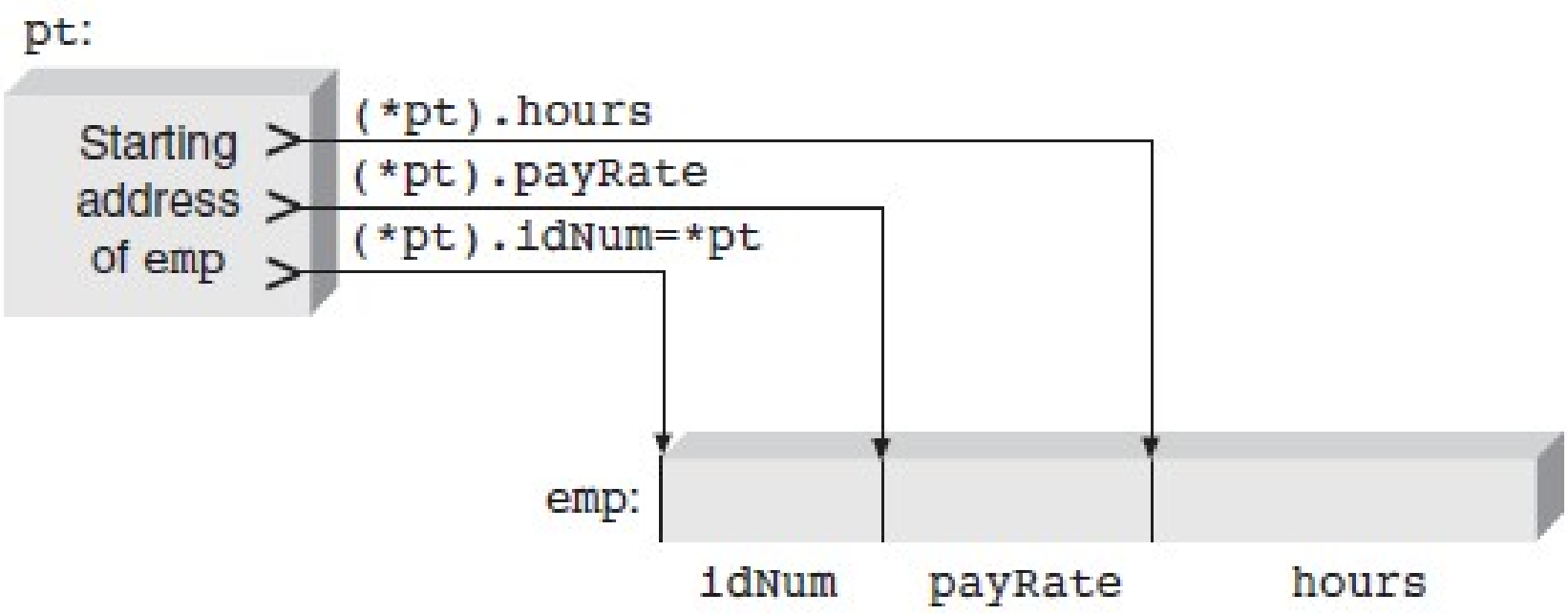
- Change Function Definition

```
30  double calcNet(Employee *pt)
31  {
32      pt->payRate++; //salary increment by the government!!
33      return (pt->payRate * pt->hours);
34  }
```

```
"C:\Users\Tinsae\Desktop\cpp scrap\lexample6_2.exe"
The net pay for employee 6782 is $402.16
Is emp payRate changed ?
Yes
```

Passing a Pointer

ASTU



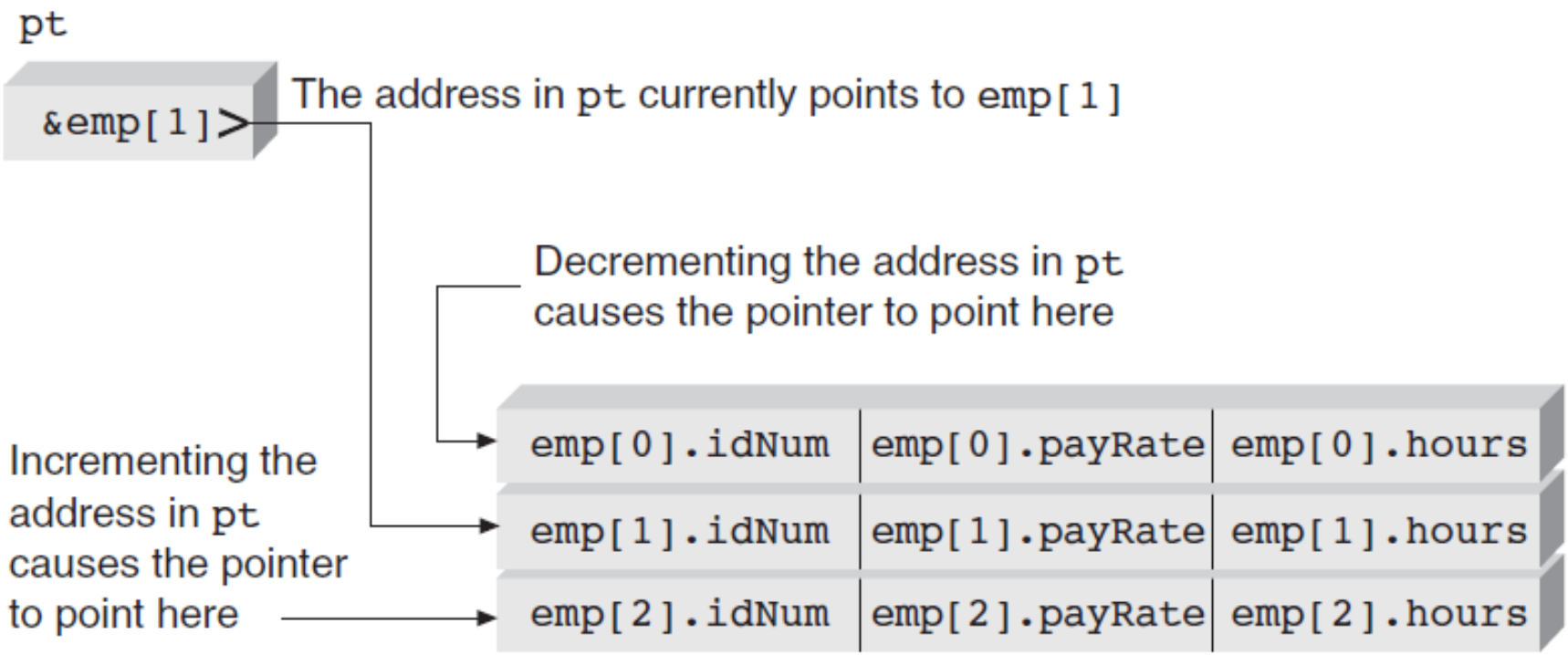
Passing a Pointer

- Using pointers to functions is very comm on
- Special notation exists for locating a me mber of a structure from a structure poi nter
  - Notation is `pointer->member`
    - Equivalent to `(*pointer).member`
  - Examples:

`(*pt).idNum`	can be replaced by	`pt->idNum`
`(*pt).payRate`	can be replaced by	`pt->payRate`
`(*pt).hours`	can be replaced by	`pt->hours`

Changing Pointer Address

ASTU



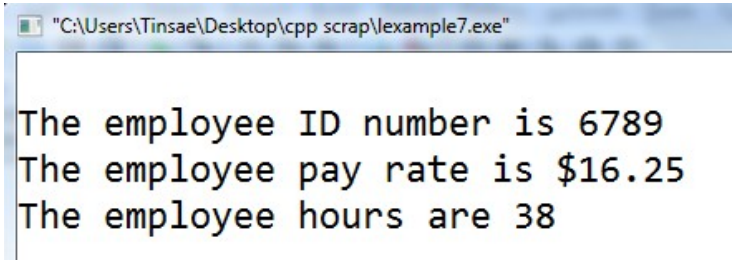
Returning Structures

- In practice, most structure-handling functions get direct access to a structure by receiving a structure reference or address
  - Changes to a structure can be made directly from a function
- To have a function return a separate structure, follow same procedure as for returning scalar values

Example 7: Returning Structures

ASTU

```
1  #include <iostream>
2  #include <iomanip>
3  using namespace std;
4  struct Employee // declare a global data type
5  {
6      int idNum;
7      double payRate;
8      double hours;
9  };
10 Employee getVals();  // function prototype
11 int main()
12 {
13     Employee emp;
14     emp = getVals();
15     cout << "\nThe employee ID number is " << emp.idNum
16          << "\nThe employee pay rate is $" << emp.payRate
17          << "\nThe employee hours are " << emp.hours << endl;
18     return 0;
19 }
20 Employee getVals() // return an Employee structure
21 {
22     Employee next;
23     next.idNum = 6789;
24     next.payRate = 16.25;
25     next.hours = 38.0;
26     return next;
27 }
```



Linked Lists

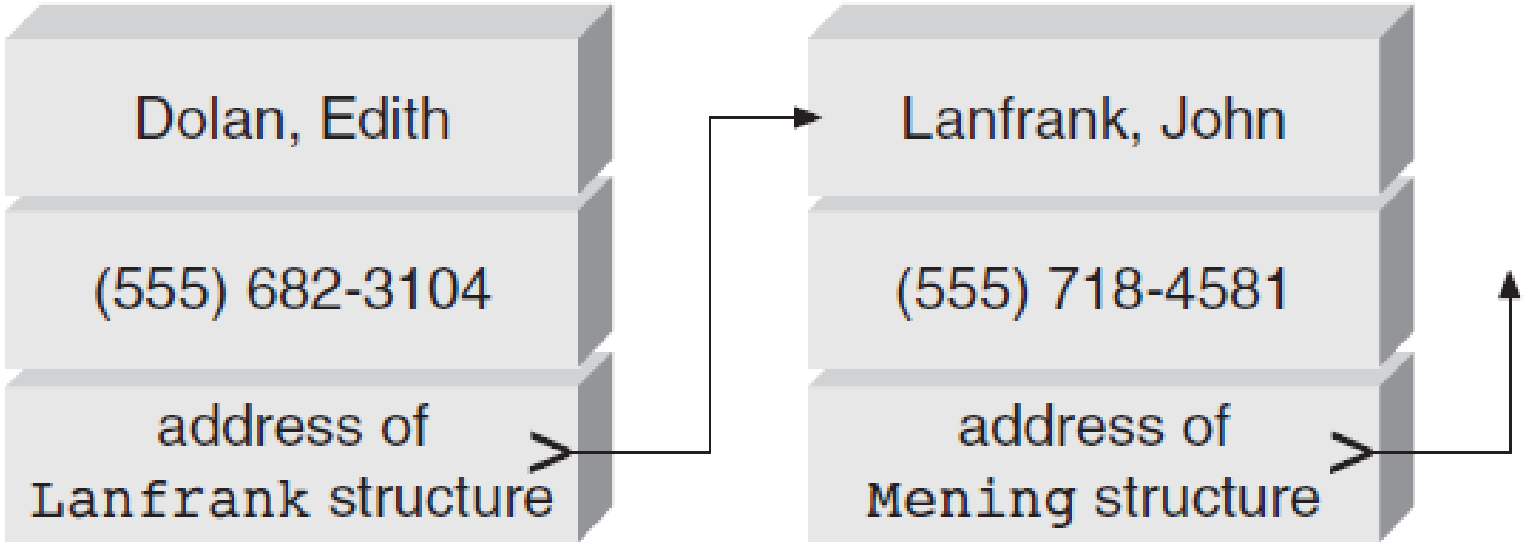
- A classic data handling problem is making additions or deletions to existing structures that are maintained in a specific order
- Linked lists provide method for maintaining a constantly changing list
- **Linked list:** Set of structures in which each structure contains at least one member whose value is the address of next logically ordered structure in list

Linked Lists

- Instead of requiring each record to be p hysically stored in correct order, each ne w structure is physically added wherever computer has free storage space
- Records are “linked” by including address of next record in the record immediately preceding it
- Current record contains address of next record no matter where next record is st ored

Linked Lists

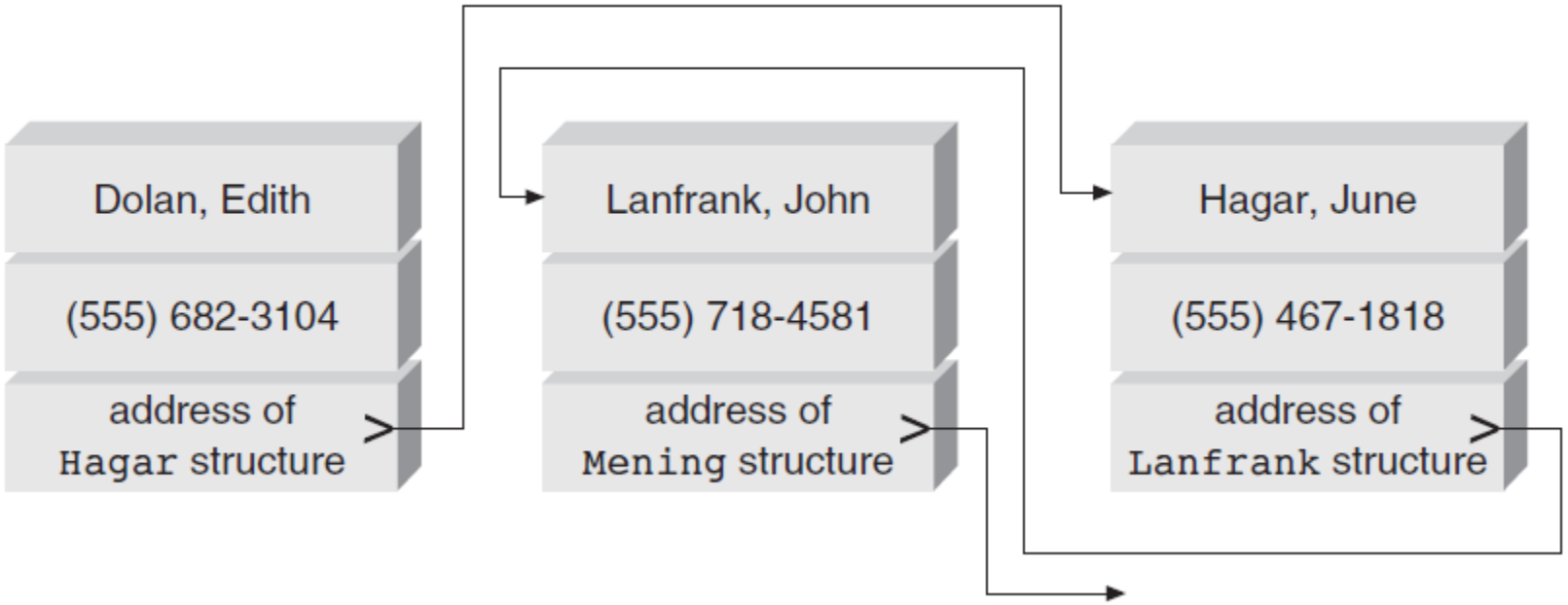
ASTU



Linked Lists

ASTU

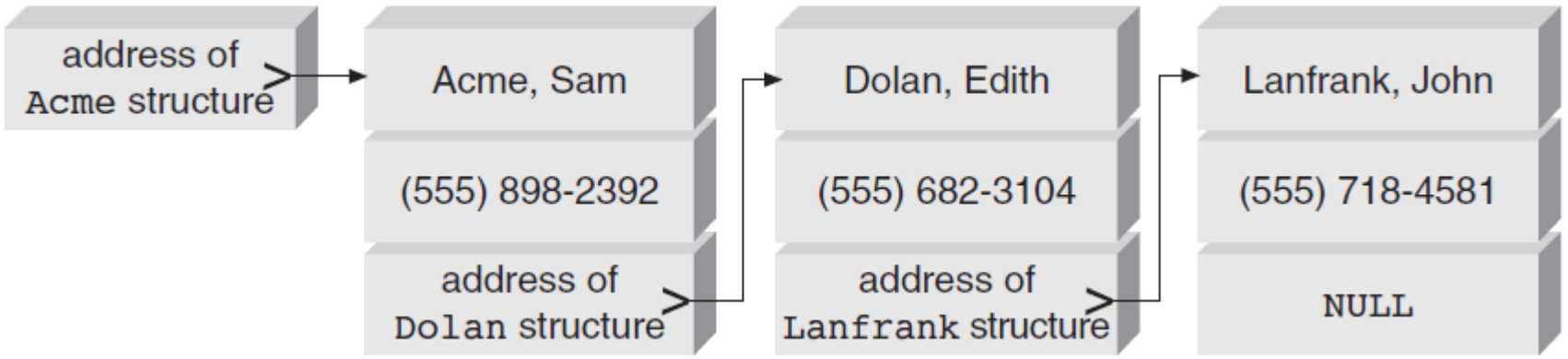
- Adjusting addresses to point to the corre ct structures



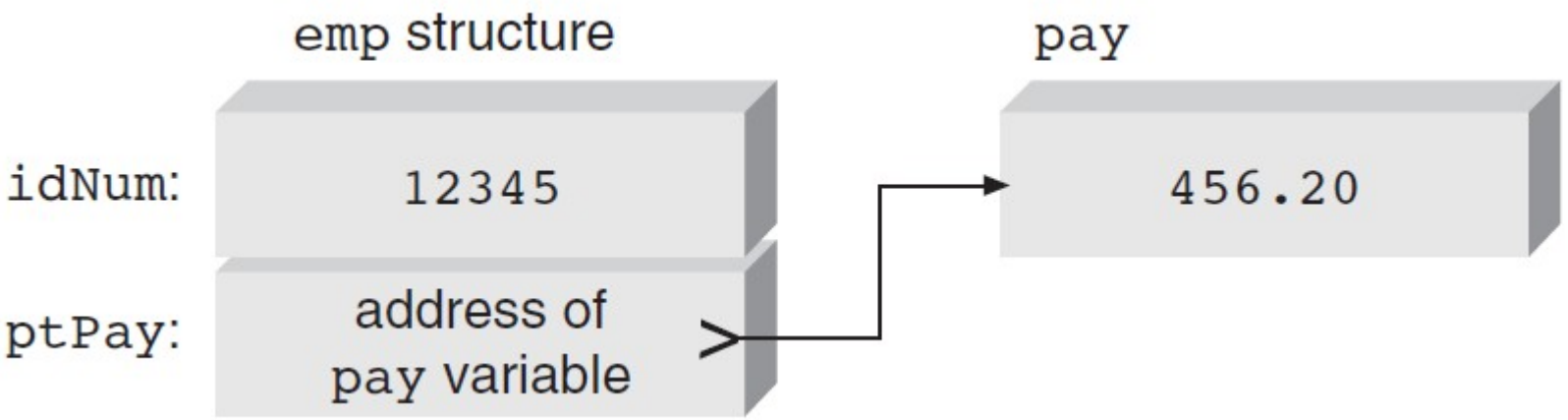
Linked Lists

ASTU

- Using initial & final pointer variables



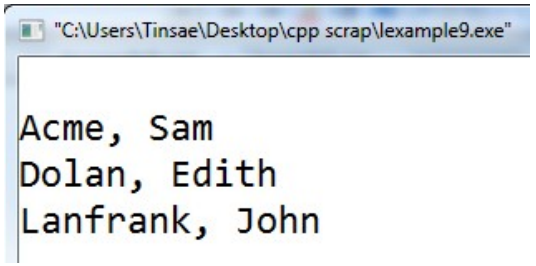
- Storing an address in a structure membe r



Example 8: Linked List

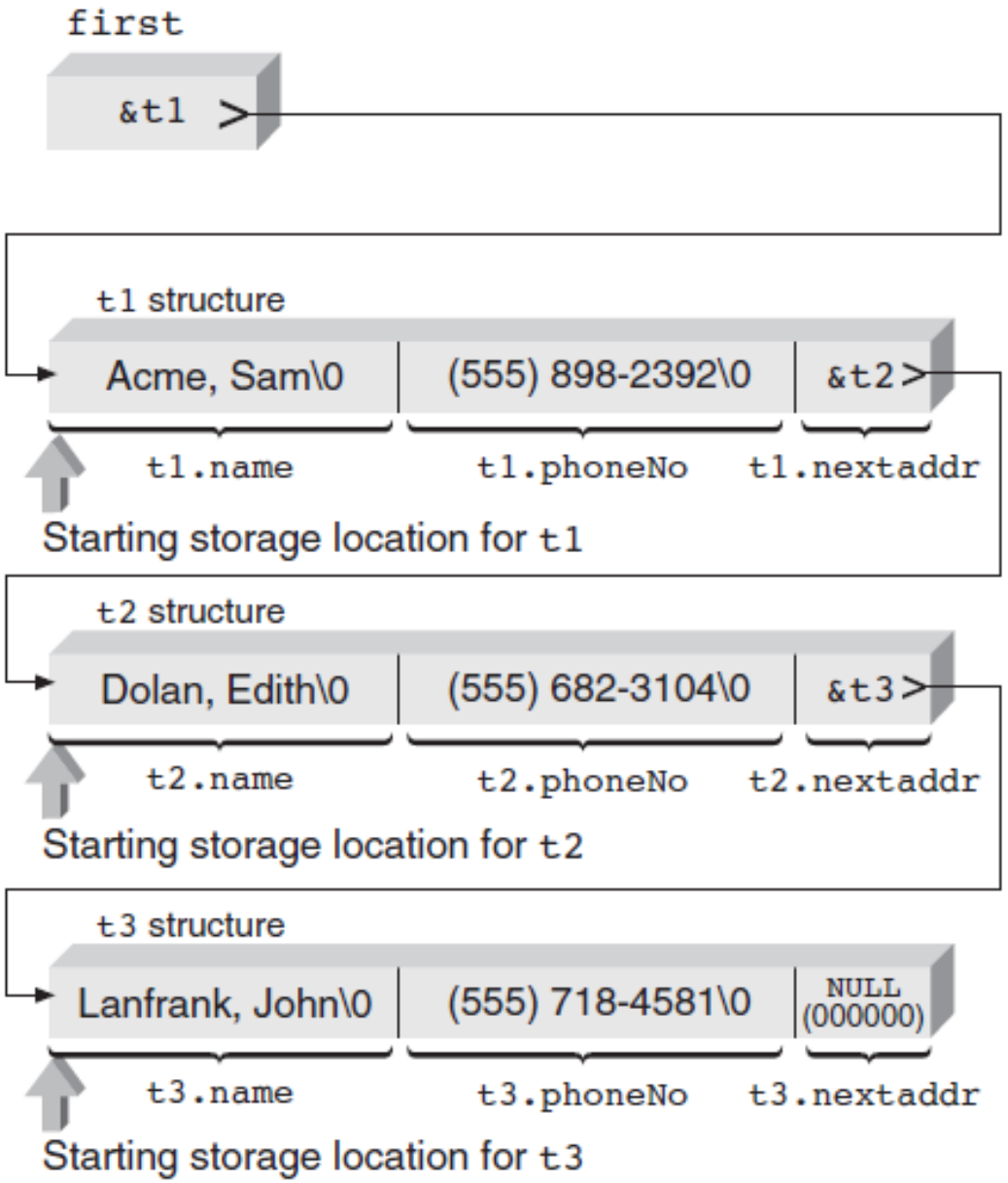
ASTU

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4  struct TeleType
5  {
6      string name;
7      string phoneNo;
8      TeleType *nextaddr;
9  };
10 int main()
11 {
12     TeleType t1 = {"Acme, Sam","(555) 898-2392"};
13     TeleType t2 = {"Dolan, Edith","(555) 682-3104"};
14     TeleType t3 = {"Lanfrank, John","(555) 718-4581"};
15     TeleType *first; // create a pointer to a structure
16     first = &t1; // store t1's address in first
17     t1.nextaddr = &t2; // store t2's address in t1.nextaddr
18     t2.nextaddr = &t3; // store t3's address in t2.nextaddr
19     t3.nextaddr = NULL; // store a NULL address in t3.nextaddr
20     cout << endl << first->name
21          << endl << t1.nextaddr->name
22          << endl << t2.nextaddr->name
23          << endl;
24     return 0;
25 }
```



The Relationship between Structures

ASTU



Example 10: Cycling through the List

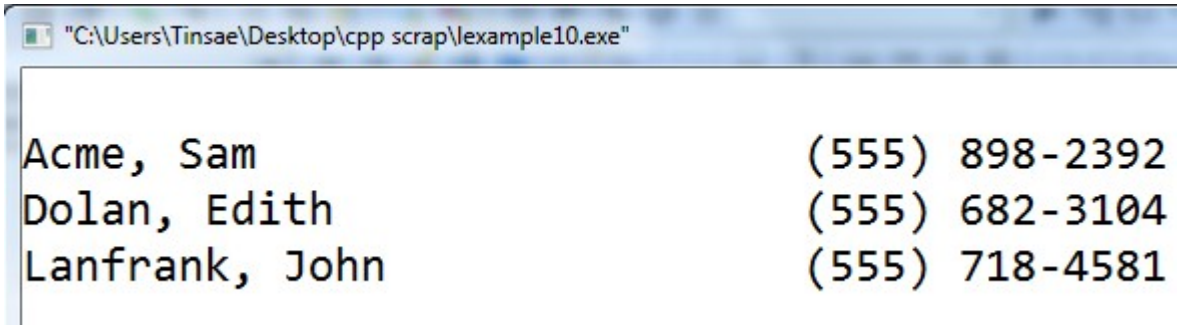
ASTU

```
#include <iostream>
#include <iomanip>
#include <string>
using namespace std;
struct TeleType
{
    string name;
    string phoneNo;
    TeleType *nextaddr;
};
void display(TeleType *); // function prototype
int main()
{
    TeleType t1 = {"Acme, Sam","(555) 898-2392"};
    TeleType t2 = {"Dolan, Edith","(555) 682-3104"};
    TeleType t3 = {"Lanfrank, John","(555) 718-4581"};
    TeleType *first; // create a pointer to a structure
    first = &t1; // store t1's address in first
    t1.nextaddr = &t2; // store t2's address in t1.nextaddr
    t2.nextaddr = &t3; // store t3's address in t2.nextaddr
    t3.nextaddr = NULL; // store the NULL address in t3.nextaddr
    display(first); // send the address of the first structure
    return 0;
}
```

Example 10: Cycling through the List

ASTU

```
25  void display(TeleType *contents)  // contents is a pointer to a
26  {
27      // structure of type TeleType
28      while (contents != NULL) // display until end of linked list
29      {
30          cout << endl << setiosflags(ios::left)
31               << setw(30) << contents->name
32               << setw(20) << contents->phoneNo ;
33          contents = contents->nextaddr; // get next address
34      }
35      cout << endl;
36      return;
37  }
```



Classes

- A procedural program consists of one or m ore algorithms that have been written in co mputer-readable language
  - Input and display of program output take a bac k seat to processing
    - Clear emphasis on formulas and calculations
- An object-oriented approach fits graphicall y windowed environments
- Abstract data types: Central to creation of objects; a user defined rather than built-in data type

Abstract Data Types

- Data type: Combination of data and ass ociated operations
- A data type defines both the types of da ta and the types of operations that can b e performed on the data
  - Data type = Allowable Data Values + Operational Capabilities
- Operations in C++ are an inherent part o f each data type

- C++ Built-in Data Type Capabilities

Capability	Example
Define one or more variables of the data type	`int a, b;`
Initialize a variable at definition	`int a = 5;`
Assign a value to a variable	`a = 10;`
Assign one variable's value to another variable	`a = b;`
Perform mathematical operations	`a + b`
Perform relational operations	`a > b`
Convert from one data type to another	`a = int(7.2);`

Abstract Data Types

- **Abstract data type (ADT):** User defined type that specifies both a type of data and the operations that can be performed on it
  - User defined types are required when you want to create objects that are more complex than simple integers and characters
- **Data structure:** How data is stored
- **Class:** C++ name for an abstract data type

Class Construction

ASTU

- A class is usually constructed in two part s:
  - **Declaration section**
    - Declares both the data types and functions for th e class
  - **Implementation section**
    - Defines the functions whose prototypes have be en declared in the declaration section

```
// declaration section
class className    // a class name is required
{
  private:
    // variable declarations
  public:
    // function prototypes
};   // end of class declaration

// implementation section
  // function definitions
```

- **Class members:** Both the variables and th e functions listed in the declaration secti on
- **Data members** or **instance variables:** Vari ables listed in the declaration section
- **Member functions**: Functions listed in th e declaration section
- When a function is part of a class it is ref erred to as a **method** to denote class me mbership

Example 1: Class Construction

ASTU

```
2   // declaration section
3   class Complex
4   {
5   private: // notice the colon after the keyword private
6       double realPart; // data member
7       double imaginaryPart; // data member
8   // function prototypes
9   public: // again, notice the colon after the keyword public
10      Complex(double = 0.0, double = 0.0); // member function (constructor)
11      void assignNewValues(double, double);  // member function
12      void showComplexValues();  // member function
13  }; // end of class declaration
14  // implementation section
15  Complex::Complex(double real, double imag)
16  {
17      realPart = real;
18      imaginaryPart = imag;
19  }
20  void Complex::assignNewValues(double real, double imag)
21  {
22      realPart = real;
23      imaginaryPart = imag;
24  }
25  void Complex::showComplexValues()
26  {
27      char sign = '+';
28      if (imaginaryPart < 0) sign = '-';
29      cout << "The complex number is "
30           << realPart << ' ' << sign << ' ' << abs(imaginaryPart) << "i\n";
31  }
```

Class Construction

- Initial uppercase letter in class name Date not required but followed by convention
- Keywords `public` and `private` are access specifiers that define access rights
  - `private:` Indicates that class member can only be accessed by class functions
- Restricting user from access to data storage implementation details is called **data hiding**
- After a class category like `private` is designated, it remains in force until a new category is specified

- `public` functions can be called from out side the class
- In general, all class functions should be `public` so that they provide capabilities to manipulate class variables from outside the class
- The function with same name as class is the class's **constructor function**
  - Used to initialize class data members with values

- Implementation section: member functions declared in the declaration section are written
- General form for functions written in the implementation section is the same as all C++ functions with the addition of the class name and the scope resolution operator ::

Example 1: Complete Program

ASTU

- Modify Example 1
  - Insert the following

```
1  #include <iostream>
2  #include <cmath>
3  using namespace std;
```

  - Add the main function

```
35 int main()
36 {
37     Complex a, b, c(6.8,9.7); // declare 3 objects
38 // Assign new values to object b's data members
39     b.assignNewValues(5.3, -8.4);
40     a.showComplexValues(); // display object a's values
41     b.showComplexValues(); // display object b's values
42     c.showComplexValues(); // display object c's values
43     return 0;
44 }
```

```
"C:\Users\Tinsae\Desktop\cpp scrap\ComplexNumbers\bin\Debug\ComplexNumbers.exe"
The complex number is 0 + 0i
The complex number is 5.3 - 8.4i
The complex number is 6.8 + 9.7i
```

- Variables of a user-declared class must:
  - Be defined before use in a program
  - Are referred to as **objects**
- An object name's attribute is referenced with the **dot operator**
  - *`objectName.attributeName`*

    `objectName` is the name of a specific object

    `attributeName` is the name of a data member defined for the object's class

- The syntax for referring to an object's m ethod is:
  - *objectName.methodName(parameters)*
    - *objectName* is the name of the specific object
    - *methodName* is name of a function defined for t he object's class

Terminology

- **Class**: Programmer defined data type from w hich objects can be created
- **Objects**: Created from classes
  - Referred to as **instances** of a class
- Process of creating a new object is called **inst antiation** of the object
- Each time a new object is instantiated a new set of data members belonging to the object is created
  - Values contained in these data members determi ne the object's **state**

Basic Class Functions

- Constructor: A function used to initialize an object's data members when the obje ct is created
- Accessor: A function that reports inform ation about an object's state
- Mutator: A function that modifies the va lues stored in an object's data members

Constructor Functions

- A **constructor function** is any function with th e same name as its class
- Multiple constructors can be defined for each class as long as they can be distinguished by n umber and types of their parameters
- A constructor's intended purpose is to initializ e a new object's data members
- If no constructor function is written, the com piler supplies a default constructor
- In addition to initialization, a constructor can perform other tasks when it is called

Constructor Functions

ASTU

- General format of a constructor includes :
  - The same name as the class to which it belo ngs
  - No return type (not even `void`)
  - A constructor that does not require argume nts is called the **default constructor**

```
className::className(parameter list)
{
  // function body
}
```

Calling Constructors

- Constructors are called when an object is created
- Declaration can be made in a variety of ways

```
Date c(4,1,2013);
Date c = Date(4,1,2013);
Date c = 8; // similar to above with
       // defaults for day and                    // y
  ear
```

- An object should never be declared with empty parentheses

```
Date a();
```

  - Not the same as the declaration `Date a;`
  - Does not result in an object being created

Example 2: Constructors

ASTU

```
1  #include <iostream>
2  using namespace std;
3  // declaration section
4  class Complex
5  {
6  private:
7      double realPart; // declare realPart as a double variable
8      double imaginaryPart;  // declare imaginaryPart as a double variable
9  public:
10     Complex(double = 0.0, double = 0.0); // constructor prototype
11 // with default arguments
12 };
13 // implementation section
14 Complex::Complex(double real, double imag) // constructor
15 {
16     realPart = real;
17     imaginaryPart = imag;
18     cout << "Created the new complex number object "
19          << realPart << " + " << imaginaryPart << "i\n";
20 }
21 int main()
22 {
23     Complex a; // declare an object
24     Complex b(6.8, 9.7);  // declare an object
25     return 0;
26 }
27
```

Accessor Functions

ASTU

- An **accessor function** provides a means f or reporting on an object's state
- Conventionally called **get()** functions
- Each class should provide a complete set of accessor functions
- Accessor functions are extremely import ant because they provide a means of retr ieving and displaying an object's private data values

```
double getReal() {return realPart;}              // inline accessor
double getImaginary() {return imaginaryPart;}    // inline accessor
```

Mutator Function

ASTU

- A **mutator function** provides a means for changing an object's data member
- Conventionally called **set()** functions
- A class can contain multiple mutators, as long as each one has a unique name or parameter list

```
void setReal(double rl) {realPart = rl;}              // inline mutator
void setImaginary(double im) {imaginaryPart = im;}    // inline mutator
```

Example 3: Accessors and Mutators

ASTU

```
1  #include <iostream>
2  using namespace std;
3  // declaration section
4  class Complex
5  {
6  private:
7      double realPart;
8      double imaginaryPart;
9  public:
10     //inline constructor
11     Complex(double real = 0.0, double imag = 0.0)
12     {
13         realPart = real;
14         imaginaryPart = imag;
15     }
16     //inline functions(without prototype)
17     double getReal()
18     {
19         return realPart;    // inline accessor
20     }
21     double getImaginary()
22     {
23         return imaginaryPart;    // inline accessor
24     }
25     void setReal(double rl)
26     {
27         realPart = rl;    // inline mutator
28     }
```

Example 3: Accessors and Mutators

ASTU

```
29      void setImaginary(double im)
30      {
31          imaginaryPart = im;    // inline mutator
32      }
33  }; // end of class declaration
34  int main()
35  {
36      Complex num1(16.4,18.9); // declare a Complex object
37
38      cout << "The real part of num1 is "
39           << num1.getReal() << endl;
40      cout << "The imaginary part of num1 is "
41           << num1.getImaginary() << "\n\n";
42      num1.setReal(25.2);
43      num1.setImaginary(-27.8);
44      cout << "The real part of num1 is now "
45           << num1.getReal() << endl;
46      cout <<  "The imaginary part of num1 is now "
47           << num1.getImaginary() << endl;
48      return 0;
49  }
```

```
"C:\Users\Tinsae\Desktop\cpp scrap\ComplexNumbers\accessormutator.exe"
The real part of num1 is 16.4
The imaginary part of num1 is 18.9

The real part of num1 is now 25.2
The imaginary part of num1 is now -27.8
```

Adding Class Functions

- Most classes typically require additional fu nctions
- In C++, there are two basic means of suppl ying these additional capabilities:
  - Construct class functions in a similar manner a s mutator and accessor functions
  - Construct class functions that use conventiona l operator symbols, such as =, *, ==, >=, which are known as operator functions **(not discussed here)**
- Both approaches can be implemented as member or friend functions

Member Functions

- Member functions can be added to a class by including their prototypes in the declaration section and providing code for the function in the implementation section or as an inline function
- The general syntax for each function header is:

```
returnType className::functionName(parameter list)
```

Example 4: Member Functions

ASTU

```
#include <iostream>
#include <cmath>
using namespace std;
// declaration section
class Complex
{
private:
    double realPart; // declare realPart as a double variable
    double imaginaryPart;  // declare imaginaryPart as a double variable
public:
    Complex(double real = 0.0, double imag = 0.0) // inline constructor
    {
        realPart = real;
        imaginaryPart = imag;
    }
    void showComplexValues(); // accessor prototype
    void assignNewValues(double real, double imag)  // inline mutator
    {
        realPart = real;
        imaginaryPart = imag;
    }
    Complex multScaler(double = 1.0);
}; // end of class declaration
void Complex::showComplexValues() // accessor
{
    char sign = '+';
    if (imaginaryPart < 0) sign = '-';
    cout << realPart << ' ' << sign << ' ' << abs(imaginaryPart) << 'i';
}
```

Example 4: Member Functions

ASTU

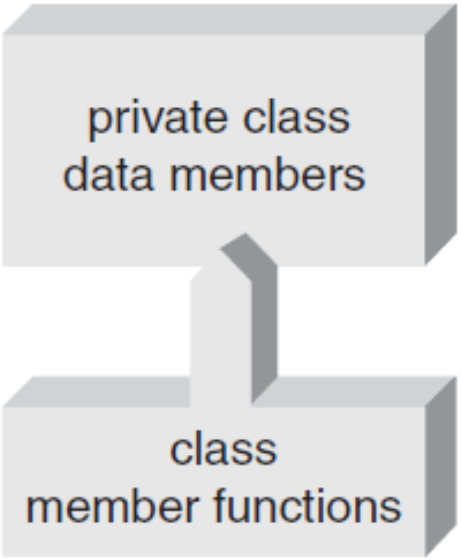
```
30  Complex Complex::multScaler(double factor)
31  {
32      Complex newNum;
33      newNum.realPart = factor * realPart;
34      newNum.imaginaryPart = factor * imaginaryPart;
35      return newNum;
36  }
37  int main()
38  {
39      Complex complexOne(12.57, 18.26), complexTwo; // declare two objects
40      cout << "The value assigned to complexOne is ";
41      complexOne.showComplexValues();
42      complexTwo = complexOne.multScaler(10.0); // call the function
43      cout << "\nThe value assigned to complexTwo is ";
44      complexTwo.showComplexValues();
45      return 0;
```

```
"C:\Users\Tinsae\Desktop\cpp scrap\ComplexNumbers\additonfunctions.exe"
The value assigned to complexOne is 12.57 + 18.26i
The value assigned to complexTwo is 125.7 + 182.6i
```

Friend Functions

ASTU

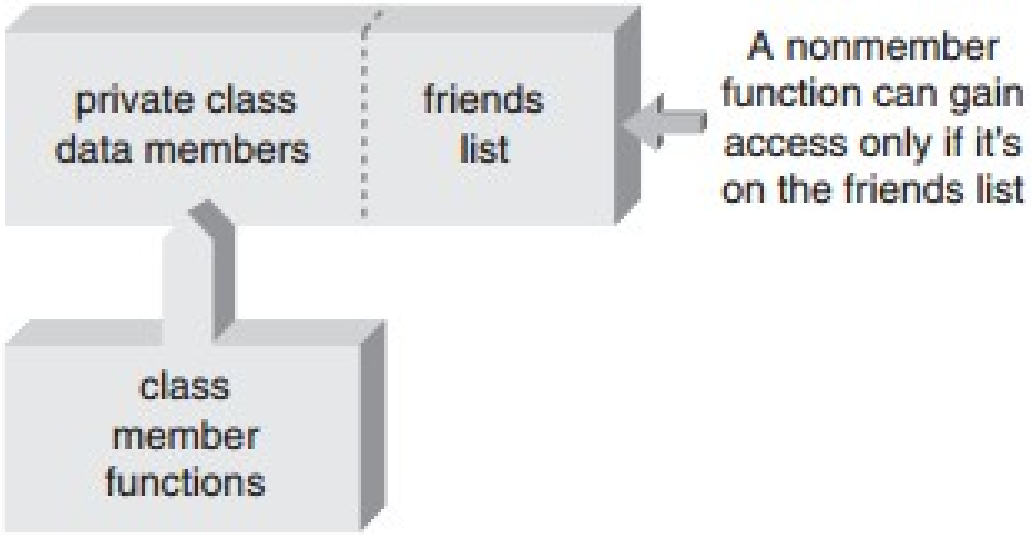
- Private variables can be accessed and ma nipulated through a class's member func tions



Friend Functions

- External access to private functions can be granted through the friends list mechanism
- The friends list members are granted the same privileges as a class's member functions
- Nonmember functions in the list are called friend functions
- Friends list: Series of function prototype declarations preceded with the `friend` keyword

Friend Functions



Example 5: Friend Functions

ASTU

```
1  #include <iostream>
2  #include <cmath>
3  using namespace std;
4  // declaration section
5  class Complex
6  {
7  // friends list
8      friend double addReal(const Complex&, const Complex&);
9      friend double addImag(const Complex&, const Complex&);
10 private:
11     double realPart;
12     double imaginaryPart;
13 public:
14     Complex(double real = 0.0, double imag = 0.0) // inline constructor
15     {
16         realPart = real;
17         imaginaryPart = imag;
18     }
19     void showComplexValues();
20 };
21 // implementation section
22 void Complex::showComplexValues()
23 {
24     char sign = '+';
25     if (imaginaryPart < 0) sign = '-';
26     cout << realPart << ' ' << sign << ' ' << abs(imaginaryPart) << 'i';
27 }
```

Example 5: Friend Functions

ASTU

```
28  // friend implementations
29  double addReal(const Complex& a, const Complex& b)
30  {
31      return(a.realPart + b.realPart);
32  }
33  double addImag(const Complex& a, const Complex& b)
34  {
35      return(a.imaginaryPart + b.imaginaryPart);
36  }
37  int main()
38  {
39      Complex a(3.2, 5.6), b(1.1, -8.4);
40      double re, im;
41      cout << "The first complex number is ";
42      a.showComplexValues();
43      cout << "\nThe second complex number is ";
44      b.showComplexValues();
45      re = addReal(a,b);
46      im = addImag(a,b);
47      Complex c(re,im); // create a new Complex object
48      cout << "\n\nThe sum of these two complex numbers is ";
49      c.showComplexValues();
50      return 0;
51  }
```

```
"C:\Users\Tinsae\Desktop\cpp scrap\ComplexNumbers\friendfriend.exe"
The first complex number is 3.2 + 5.6i
The second complex number is 1.1 - 8.4i

The sum of these two complex numbers is 4.3 - 2.8i
```

Case Study: Constructing Date Object

- Step 1: Analyze the problem: define operations
- Step 2: Develop a solution: define classes and data members
- Step 3: Code the solution
- Step 4: Test and correct the program: testing the `Date` class entails testing and verifying each class function and operator function

Case Study: Constructing Date Object

ASTU

- Data Structure
  - In this application, you have one type of obj ect, a date, which is specified by a month, d ay, and year.

Case Study: Constructing Date Object

ASTU

- Required Operations
  - A complete description of a Date class must include how a date is to be represented and the operations the class must provide.

Operation	Return Value
Initialize	None (use a constructor)
Modify	None (use a mutator)
Display	None (use an accessor)
Is the year a leap year?	Boolean
Is the date a weekday?	Boolean (use Zeller's algorithm; see Exercise 5)
Is the day a holiday?	Boolean
Are two dates the same?	Boolean
Does one date occur before a second date?	Boolean
Does one date occur after a second date?	Boolean
Determine the day of the week	Boolean (use Zeller's algorithm; see Exercise 5)
Determine the next day's date	Date
Determine the previous day's date	Date
Determine the day difference between two dates	Integer

Case Study: Constructing Date Object

ASTU

- Data Members

```
// declaration section
class Date
{
  private:
     int month;
     int day;
     int year;
};
```

- Algorithm to Determine Leap Year

If the year is divisible by 4 with a remainder of 0
AND the year is divisible by 100 with a nonzero remainder
OR the year is divisible by 400 with no remainder
then the year is a leap year
Else
the year is not a leap year
EndIf

Code the Solution

ASTU

```
1  #include <iostream>
2  #include <iomanip> // needed for formatting
3  using namespace std;
4  // declaration section
5  class Date
6  {
7  private:
8       int month;
9       int day;
10      int year;
11 public:
12      // inline constructor
13      Date(int mm = 1, int dd = 1, int yyyy = 2014)
14      {
15           month = mm;
16           day = dd;
17           year = yyyy;
18      }
19      // inline mutator
20      void setDate(int mm, int dd, int yy)
21      {
22           month = mm;
23           day = dd;
24           year = yy;
25      }
```

Code the Solution

ASTU

```
26      void showDate(); // accessor
27      bool isLeapYear();
28      bool operator==(const Date&);
29  };
30  // implementation section
31  void Date::showDate()
32  {
33      cout  << "The date is " << setfill('0')
34            << setw(2) << month << '/'
35            << setw(2) << day << '/'
36            // extract the last two year digits
37            << setw(2) << year % 100;
38      cout  << endl;
39  }
40  bool Date::isLeapYear()
41  {
42      if ((year % 4 == 0 && year % 100 != 0)
43              || (year % 400 == 0))
44          return true; // is a leap year
45      else
46          return false; // is not a leap year
47  }
```

- Testing is left as an exercise

Structures Versus Classes

- Structures are normally used with all member variables public and with no member functions.
  - However, in C++ a structure can have private member variables and both public and private member functions
  - Aside from some notational differences, a C++ structure can do anything a class can do.
- Having said this and satisfied the “truth in advertising” requirement, we advocate that you forget this technical detail about structures.

Structures Versus Classes

- If you take this technical detail seriously an d use structures in the same way that you u se classes,
  - then you will have two names (with different s yntax rules) for the same concept.
- On the other hand, if you use structures as we described them, then you will have a m eaningful difference between structures (as you use them) and classes; and your usage will be the same as that of most other prog rammers.

- One difference between a structure and a class is that
  - they differ in how they treat an initial group of members that has neither a public nor a p rivate access specifier.
  - If the first group of members in a definition i s not labeled with either public :or private: ,t hen a structure assumes the group is public, whereas a class would assume the group is private.