CS 280
Programming Language Concepts

Spring 2025

C++ Classes

Topics

- **Record Types in C/C++: struct and union**
- **C++ Classes**
- **Nested Classes in C++**
- **More about Constructors and Destructors**
- **Class Interface and Implementation**
- **C++ Class Templates**

Record Types in C/C++

- A *record* is a possibly heterogeneous aggregate of data elements in which the individual elements, called fields, are identified by names.
- struct: a collection of elements of different types
  - In C++, structures are similar to classes. The fields are called members.

```
struct employeeType{ //declaration of a data type
    int id;
    char name[25];
    float salary;
};
struct employeeType employee;//in C definition variable
employeeType employee;//in C++ definition of variable
...
employee.salary = 5545;//access of member using dot notation
```

Record Types in C/C++

- Union: A union is like a struct in that it generally has several fields, all of which are public by default. Unlike a struct, however, only one of the fields is used at any given time. In other words, it is a structure that allows the same storage space to be used to store values of different data types at different times.
  - A union is a special class type that can hold only one of its non-static data members at a time.
  - Example:

```
union UnionTyp {
        int i;
        double d;
        char c;
    };
    UnionType u;
u.i = 5;
. . . .
u.c ='M';
```

C++ Classes

- Based on C **`struct`** type and Simula 67 classes.
- The class is the encapsulation device.
- A class is a type.
- All of the class instances share a single copy of the member functions.
- Each instance of a class has its own copy of the class data members
- Instances can be static, stack dynamic, or heap dynamic.
- C++ defines a struct to be a class whose members are all, by default, public. Furthermore, it is common practice to use only data as members of a struct.
- Information Hiding
  - *Private* clause for hidden entities
  - *Public* clause for interface entities
  - *Protected* clause for inheritance (Chapter 12)

C++ Classes (continued)

```
class Stack {
   private: //members that are visible only to other members or friends
        int *stackPtr, maxLen, topSub;
   public: //Members that are visible to clients
        Stack() { // a constructor
                stackPtr = new int [100];
                maxLen = 99;
                topSub = -1;
        };
        ~Stack () {delete [] stackPtr;};  //destructor

        void push (int number) {
           if (topSub == maxLen)
             cerr << "Error in push - stack is full\n";
           else stackPtr[++topSub] = number;
        };
```

C++ Classes (continued)

```
    void pop () {
       if(empty())
            cerr << "Error in pop—stack is empty\n";
       else topSub--;
    }
    int top () {
       if (empty())
             cerr << "Error in top—stack is empty\n";
       else
            return (stackPtr[topSub]);
    }
    int empty () {
        return (topSub == -1);
    }
}
```

C++ Classes (continued)

```
void main(){
  int topOne;
  Stack stk; //create an instance of stack class
  stk.push(42);
  stk.push(17);
  topOne = stk.top();
  stk.pop();
  . . .
}
```

C++ Classes (continued)

- Constructors:
  - Functions to initialize the data members of instances (they *do not* create the objects)
    - There may be more than one constructor defined for a class.
    - A constructor that can be called with no arguments is known as a default constructor.
  - May also allocate storage if part of the object is a pointer to a dynamically created object or data structure.
  - Can include parameters to provide parameterization of the objects
  - Implicitly called when an instance is created
  - Can be explicitly called
  - Name is the same as the class name

C++ Classes (continued)

- Destructors
  - Functions to cleanup before an instance is destroyed; usually just to reclaim heap storage
  - Implicitly called when the object's lifetime ends
  - Can be explicitly called
  - Name is the class name, preceded by a tilde (~)
  - If *no destructor* is specified, destructor *automatically provided* that calls destructor for each data member of class type, and performs no clean-up action.

Example: Class `Time`

```
class Time {
public:
   Time( int = 0, int = 0, int = 0 );
      // default constructor with default parameters
   void setTime( int, int, int ); //set hr, min,sec
   void printMilitary();      // print am/pm format
   void printStandard();     // print standard format

private:
   int hour;
   int minute;
   int second;
};
```

Declaring Time Objects

```
// Note that implementation of class Time not given
// here.


int main(){
   Time t1,     // all arguments defaulted
        t2(2), // min. and sec. defaulted
        t3(21, 34),   // second defaulted
        t4(12, 25, 42); // all values specified
   //dynamically created object
   Time * tPtr = new Time(12, 25, 42);
     . . .
}
```

```
class ArrayDouble { // array of doubles class
public:
  ArrayDouble (); // create empty array
  ArrayDouble (int size ); // create array of specified size
  // ...
private:
  // ...
};
class Vector { // n-dimensional real vector class
public:
  Vector(int size, int y) : data_(size), num(y) {}
  // force data_ to be constructed with
  // ArrayDouble::ArrayDouble(int)
  // ...
private:
  ArrayDouble data_; // elements of Vector
  int num;
};
```

Initialization Lists

- Using initialization lists to initialize parameters instead of assignment statements in the body saves time when data members are complex classes
- Using initialization lists to initialize constant (const) parameters is required.
- If the data member in a class that does not have a zero-parameter constructor then it must be initialized using initialization list.
- Initializer list forces specific constructors to be used to initialize individual data members before body of constructor is entered
  - data members always initialized in *order of declaration*, regardless of order in initializer list

Class Interface and Implementation

- In C++, separating the class interface from its implementation is common.
  - The interface remains the same for a long time.
  - The implementations can be modified independently.
  - The writers of other classes and modules have to know the interfaces of classes only.
- The interface lists the class and its members (data and function prototypes) and describes what can be done to an object.
- The implementation is the C++ code for the member functions.

Separation of Interface and Implementation

- It is a good programming practice for large-scale projects to put the interface and implementation of classes in different files.
  - For small amount of coding it may not matter.
- *Header File*: contains the interface of a class. Usually ends with `.h` (an include file)
- *Source-code file*: contains the implementation of a class. Usually ends with `.cpp` (`.cc` or `.C`)
  - .cpp file includes the .h file with the preprocessor command `#include`.
    - Example: `#include "myclass.h"`

Separation of Interface and Implementation

- A big complicated project will have files that contain other files.
  - There is a danger that an include file (.h file) might be read more than once during the compilation process.
    - It should be read only once to let the compiler learn the definition of the classes.
- To prevent a `.h` file to be read multiple times, we use preprocessor commands `#ifndef` and `#define` in the following way.

Class Interface

```
#ifndef  _IntCell_H_
#define _IntCell_H_

class  IntCell
{
   public:
      IntCell( int initialValue = 0 );
   int read( ) const;
   void write( int x );
   private:
   int storedValue;
};
#endif
```

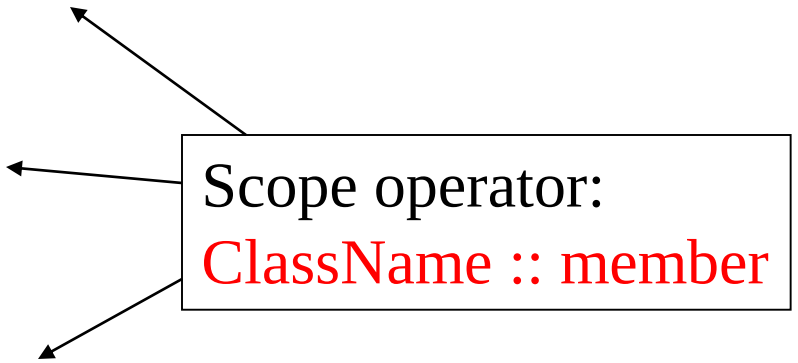
`IntCell` class Interface in the file *IntCell.h*

Class Implementation

```
#include <iostream>
#include "IntCell.h"
using std::cout;

//Construct the IntCell with initialValue
IntCell::IntCell( int initialValue)
   : storedValue{ initialValue} {}

//Return the stored value.
int IntCell::read( ) const
{
    return storedValue;
}
//Store x.
void IntCell::write( int x )
{
    storedValue = x;
}
```



`IntCell` class implementation in file *IntCell.cpp*

Nested Classes in C++

```
class single_Linked_List {
        private:
                class node{
                        public:
                                node *link;
                                int contents;
                };
                node *head;
        public:
                single_Linked_List() {head = 0;
                }
                void insert_at_head(int);
                void insert_at_tail(int);
                int remove_at_head();
                int empty();
};
```

Nested Classes in C++

- A class can be nested in every part of the surrounding class: in the public, protected or private section.
- To grant the surrounding class access rights to the private members of a nested class the nested class may declare its surrounding class as a friend.
- If a member of the surrounding class should use a (non-static) member of a nested class then the surrounding class must define a nested class object, which can thereupon be used by the members of the surrounding class to use members of the nested class.

C++ Class Templates

- Class template is a family of classes parameterized on one or more parameters
- Syntax has general form:

```
template <parameter list> class
```

  - parameter list: parameter list for class is a sequence of one or more parameter declarations separated by commas, where each is defined as of either **class** identifier or **typename** identifier
    - The list does not mean that the list of items must be objects of a C++ class. They can hold any data type that we specify, including built-in or user defined types.
    - class: class/struct declaration or definition

C++ Class Templates

- Example:

```
// declaration of class template
template <class T, unsigned int size >
class MyArray;
// definition of class template
template <class T, unsigned int size >
class MyArray {
// ...
T array_[size ];
};
MyArray <double, 100> x;
//class instantiation and Object creation
```

C++ Class Templates

```
Template <class T>
class Complex{
   private:
    T re, im;
  public:
    Complex( T x = 0, T y = 0 );
    ~Complex();
    Complex operator* ( Complex & rhs ) const;
    T getRe() const;
    T getIm() const;
    void setRe(T x);
    void setIm(T x);
    T modulus() const;
    friend ostream & operator<< (ostream &os, Complex<T> &
   rhs);
}
Complex <int> c1;
Complex <double> c2
```

C++ Class Templates

- When the compiler instantiates a template, it literally substitutes the template argument for the template parameter throughout the class template.
- Class template parameters can have *default values*
- example:

```
template <class T = int, unsigned int size = 2>
struct MyArray {
T data[size ];
};
MyArray <> a; // MyArray<int, 2>
MyArray <double> b; // MyArray<double, 2>
MyArray <double, 10> b; // MyArray<double, 10>
```

C++ Class Templates

- Definition of class member functions in an implementation file:
  - They need to be defined as **function templates** so that the compiler can associate each one with the proper template class.
  - For example

```
template <class T>
T Complex <T>::getRe() const{
    return re;
}
template <class T>
void Complex <T>::setRe(T x) const{
    re = x;
}
```