CS 225 Data Structures

Lab 2: Introduction to C++ / Object Orientation

Klaus-Dieter Schewe

ZJU–UIUC Institute, Zhejiang University

International Campus, Haining, UIUC Building, B404

email: kd.schewe@intl.zju.edu.cn

2 Object-Oriented Features of C++

2.1 Classes

Classes define **abstract data types** by means of data and functions members

Members can be **public** or **private**—only public members can be accessed from outside the scope of a class, whereas private members are encapsulated inside the class

Declarations of data and function members are done in the same way as in the imperative core

However, it is advisable to separate the definition of functions (including inline functions) from the class declaration using the **scope resolution operator ::**

Objects of a class are defined in the same way as for data types using **class_name object_name;**

Members of an object can be accessed by the the **class member access operator**, i.e. by using expressions of the form **object_name.member_name**

Example

```
class int_list {
public:
    int_list(int size = 20);
    int get_item(int index);
    void set_item(int newitem);
    void insert(int item);
    int length;
private:
    int maxsize;
    int *data; }

inline int_list::int_list(int size)
{   maxsize = size;
    data = new int[size];
    last = -1; }
int int_list::get_item(int index)
{   if (0 <= index && index <= length)
        { return data[index]; }
```

Example (cont.)

```
    else
        { cout << "list index " << index << " out of range\n";
          return(EXIT_FAILURE); } }
void int_list::set_item(int index, int newitem)
{   if (0 <= index && index < length)
        { data[index] = newitem;
          return; }
    else
        { cout << "list index " << index << " out of range\n";
          return(EXIT_FAILURE); } }
void int_list::insert(int item)
{   if (length < maxsize)
        { data[length + 1] = item;
          ++length; }
    else
        { cout << "list is already filled\n";
          return(EXIT_FAILURE); } }
```

Classes and Functions

Objects of a class can be arguments of functions, and can be returned by functions

Example:

```
int sum(int_list list)
{ int sum = 0;
  for (i=0; i <= list.length; ++i)
  { if (sample.getitem(i) % 2 != 0) sum+= i; }
}
int main()
{ int_list sample = int_list(20);
  for (i=0; i < 20; ++i) {sample.insert(3*i*i-i+1);}
  return sum(sample);
}
```

In order to avoid redundant copying it is advisable to use a reference argument in such a function; otherwise a new object might be created by the function and copied(!) to another new object in the **return** statement

Access Operators

The c ommon access to members uses the class member access operator **object_name.member_name**

If pt is a pointer to an object of some class, then **(*pt).member** access the (public) class member that is declared in the class; this can be abbreviated by **pt->member** using another **class member access operator ->**

Using the scope resolution operator, we may use **type_name class_name::*pointer** to define a pointer to a class member of the given type, and an assignment of the form **pointer = &class_name::member** then creates a pointer to the specified class member

Then **object.*pointer** or **(object.*pointer)(...)** accesses the member of an object pointed to; **.*** is the **pointer-to-member access operator**

A second **pointer-to-member access operator** is **->***, where **pt_s->*pt** is a shortcut for **(*pt_s).*pt**

Miscellaneous

Class members can be declared to be **static**, as in **static int total;**

Static data members are by default initialised as 0, but a different initialisation is possible (though not within the class declaration, but using
type class_name::member_name = value

Static members are shared by all objects of a class; the same applies to static member functions

A call of a static member function of a class uses the class name rather than the object name

Objects of a class can also be declared as constant using the preceding keyword **const**

Functions on such constant objects can only be executed, if they also have been declared as constant (not modifying the members); in this case the keyword **const** is added at the end of the function declaration

Friend Functions and Classes

A function can be declared to be a **friend** of a class, e.g. in
friend int get_item(int index);

A friend function can access the members of the class, but it is not a member of the class itself

More generally, we can declare a class to be a friend, e.g. in **friend class graph**

Then all functions of the friend class have access to all members of the given class

It is therefore possible to define **private classes** with only private members, but with certain classes as friends

Then only the friend classes can access the members of a private class; otherwise a private class would be useless

Note that the friend concept is not transitive

Constructors and Destructors

A member function with the same name as the class itself is a **constructor function**

A class may have more than one constructor function, but the argument list must differ

For instance, **int_list(int size = 20)** in our class example above declares a constructor for the class int_list

A constructor is used to initialise a new object (as in our previous example)

We could define another constructor by **int_list(int_list &ℓ)** using the members of a specific object as the default

A **destructor function** has a name **~class_name**

A destructor has no return type, no arguments, and cannot be static

2.2 Inheritance in C++

In C++ a class may be declared as a **derived class** using **class derived_class : [public|private] base_class$_1$, . . . , [public|private] base_class$_n$ { . . . }**

There must be one or more previously defined classes from which the class can be derived; these classes can also be derived from other classes

In case $n = 1$ we talk of **simple inheritance**, otherwise of **multiple inheritance**

A derived class defines its members in the same way as any other class, but in addition has all data and function members of its base class(es); it **inherits** these members (except constructors and destructors) from the base class(es)

A base class can be **private** or **public** for a derived class (private is the default)

If a base class is private for a derived class, all inherited members are private

If a base class is public for a derived class, all inherited public members become public, but private members remain private to the base class

Dominance

In C++ the inheritance concept merely extends the class definition(s) by new members (**inclusion polymorphism**), but the same names may be overloaded (**ad-hoc polymorphism**)

If member names are re-used, any ambiguity is resolved by the member definition in the derived class being **dominant**, i.e. the class member access operator always refers to the member definition for the derived class

If member names are re-used, the member defined for a base class can still be accessed by using the scope resolution operator **::**, as in **object.base_class::member**

Here **object** is an object of a class derived from **base_class**, and **member** was defined for this base class

If member names are not re-used, it may also be necessary to use the class resolution **::**, if there is more than one base class, and member names are not unique

Ad-hoc Polymorphism

The overloading of function names with different definitions in derived and base classes is called **ad-hoc polymorphism** in type theory

It may become necessary that function members of different classes derived from a common base class are needed, but the class of an object is only known at run-time

For this C++ allows us to declare a function member (but not a constructor or destructor) to be virtual using the keyword **virtual**, which must precede the function declaration in the base class (and only there)

It is also possible not to have a function definition for the base class (only a declaration), if such a function would only make sense for specific derived classes

The function declaration in this case takes the form

virtual type function_name(. . . arguments . . .) = 0

indicating that the function is a **pure virtual function**, i.e. it is undefined

A class with with at least one such undefined member functions is called an **abstract class**

Access Specifiers

In addition to **private** and **public** C++ provides another **access specifier**: **protected**

Access specifiers are used for the declarations of class members and in the list of base class(es) of a derived class

The following rules govern the combination of these access specifiers:

- If a base class is private, all its members are private in derived class
- If a base class is public, all its members have the same access specifier in the base and any derived class
- If a base class is protected, its public and protected members are protected in any derived class, while private members remain private

Miscellaneous

As derived classes can be defined using another derived class as its base class, we can define deep class hierarchies (note that circularity is not permitted)

With multiple inheritance it is possible that a class becomes an indirect base class in multiple ways, i.e. an object may contain multiple copies of members from such an indirect base class

This can be avoided by making base classes virtual using the keyword **virtual** preceding the access specifier

Constructors and destructors cannot be inherited, but a definition of a constructor of a derived class must call the constructor of the base class:
derived_class_constructor(...) : base_class_constructor(...){ ... }

In case of multiple inheritance a list of base class constructors can be called; this list may even contain constructors of virtual indirect base classes

2.3 Parametric Polymorphism in C++

Strong typing requires that all arguments of a function are typed and a return type must be specified

In order to enable the definition of functions that depend only on restricted properties of a type, C++ provides the concept of a **function template**, which permits data types and classes as parameters

In type theory this is called **parametric polymorphism**

A template argument list is a list of class/type variables each preceded by the keyword **class**, i.e. the syntax is $\langle$**class** T_1**, . . . , class** $T_n\rangle$

A function template is then declared by **template template_argument_list function_declaration**, where in the function_declaration the class/type variables are treated just like any other type name

A function template definition extends such a declaration by a function body; again, the class/type variables are treated just like any other type name

Function Template Calls

Example. Define a min function, which only requires that a total order $\leq$ is defined for a type

```
template<class T> T min(T m, T n)
  { return ((m <= n) ? m : n) }
```

A function template can be called like any other function, e.g. `min(x,y)`

It is not always possible to omit the concrete type replacing the parameter in the template

For instance, using `min(x,y)` for an integer x and a double y could return either a double (i.e. treat x as double) or an integer (i.e. coerce y to an integer)

Such conflicts can be avoided by explicitly giving the argument type, e.g. we could use `min<double>(x,y)` or `min<int>(x,y)`

Class Templates

The template concept is not restricted to functions; it can as well be used for classes

In this case the class template declaration becomes
template template_argument_list class_declaration, where in the class_declaration the class/type variables are treated just like any other type name

Within the class template declarations and definitions there is no need to qualify the parameterised class by the type/class parameters

However, the definition of class members outside the scope of the class template declaration requires these parameters, i.e. the prefix
template template_argument_list is needed

Furthermore, the type/class parameters must be used in the function names

Examples and Exercises

The folder `AList (incomplete)` contains the template for lists based on unbounded arrays

Exercises:

1. Complement the function definitions in `AList.cpp`

2. Derive an implementation of a stack data structure (see the Figure `stack_cpp.png`)

3. Modify the implementation to an implementation of a deque data structure using unbounded circular lists: `CList.h`, `CList.cpp` emphasising functions `pushback`, `popback`, `pushfront` and `popfront`

4. Derive an implementation of a data structure for FIFO queues (see the Figure `queue_cpp.png`)

3 Miscellaneous Topics

File Organisation. A program file containing **main()** may (conditionally or unconditionally) include other files, in particular header files and libraries

Declarations of functions and classes are usually kept in header files (extension .h), while function definitions (except inline functions), class member definitions, etc. are placed in the program file (extension .cxx or .cpp depending on the environment)

Namespaces. In order to prevent name clashes (from inclusion of multiple header files) C++ permits the definition of **namespaces**

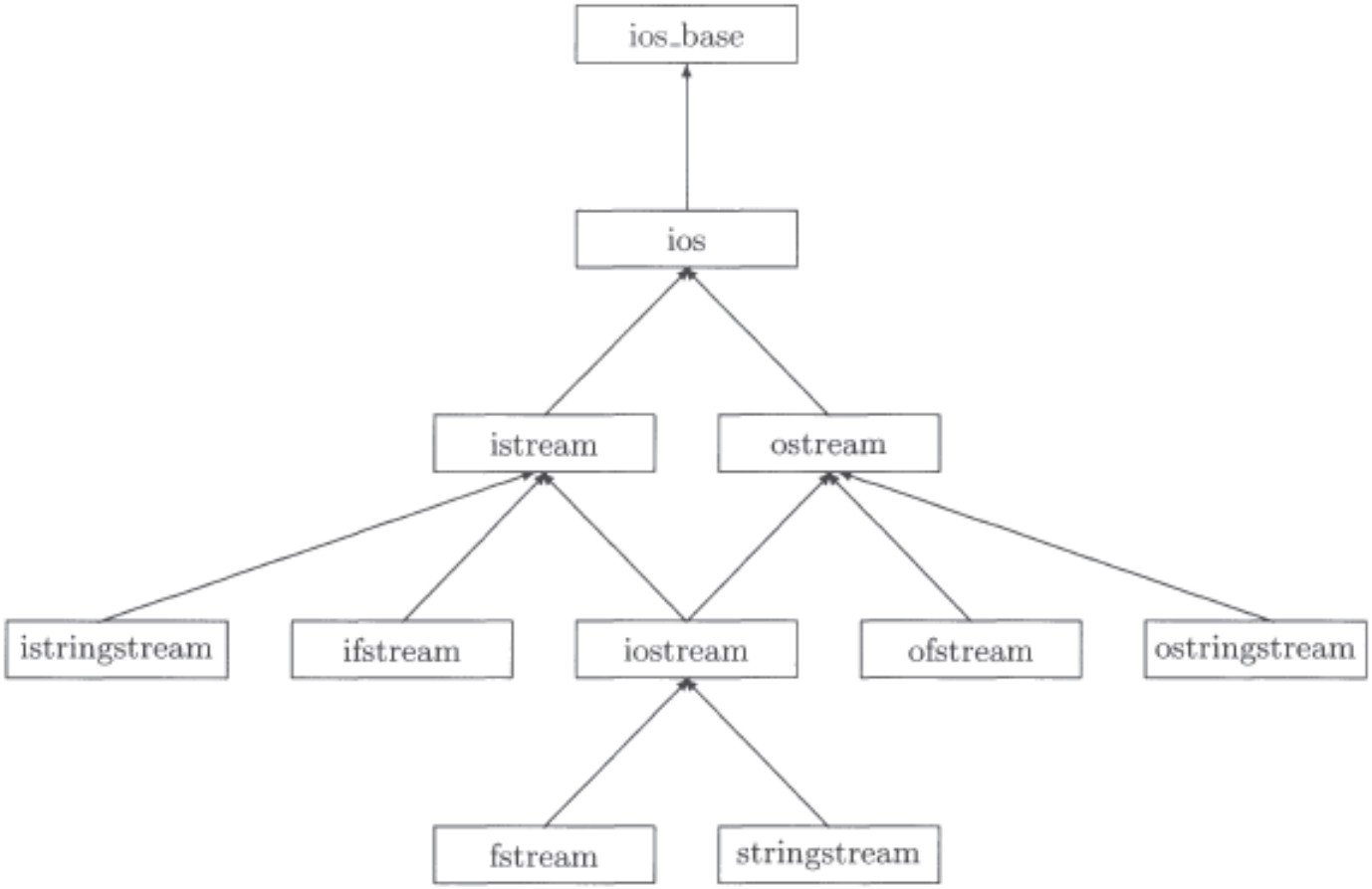
Names declared in some program or header file can be added to a named namespace

With **using namespace** or **using namespace::name** all or selected names from a namespace can be made accessible in a program or header file

Input and Output

Input and output are supposed to be taken from or directed to streams, respectively

For these a hierarchy of library header files is available



I/O Streams

For instance, ⟨iostream⟩ defines **cin** and **cout**

Input is taken from the input stream using **cin >> ...**, whereas output flows towards the output stream, i.e. we use **cout << ...**

The list of I/O library header files is the following:

Header	Purpose
`<iosfwd>`	forward declarations
`<iostream>`	standard iostream objects
`<ios>`	iostreams base classes
`<streambuf>`	stream buffers
`<istream>`	input stream template
`<ostream>`	output stream template
`<iomanip>`	formatting and manipulators
`<sstream>`	string streams
`<fstream>`	file streams
`<cstdio>`	C-style I/O

Files for Input and Output

For input from a file we need to **#include ifstream**, we output to a file analogously **#include ofstream**

With ifstream we can declare a file, e.g. "example_input.dat" to be used for the input stream:

ifstream input_file("example_input.dat");

Then the declared **input_file** take on the role of cin, i.e. we can read input from the file using

input_file >> x

Analogously, with ofstream we can declare a file, e.g. "example_output.dat" to be used for the input stream:

ifstream output_file("example_output.dat");

Then the declared **output_file** take on the role of cout, i.e. we can read input from the file using

output_file << x

For input and output to file there are several options how to open the file, how to handle errors, how to write or read data, etc. that need to be specified with the declaration of I/O files (see the C++ documentation or a C++ textbook)