

## COMP3062    Advanced Databases

### 99-02    Laboratory 1: C++

## 1    Input/Output

C++ can be considered as a better C. The minimal support for procedural programming consists of functions, arithmetic, and looping constructs, etc. In C, a program that prints out “Hello, World!” is as follows.<sup>1</sup>

```
#include <stdio.h>

main()
{
    printf("Hello, World!\n");
}
```

However, in C++, a program that does the same job looks like,

```
#include <iostream.h>

main()
{
    cout << "Hello, World!\n";
}
```

The line `#include <iostream.h>` instructs the compiler to include the declarations of the standard stream input and output facilities as found in `iostream.h`. Input and output are not a part of the C++ language, but are supported by a library written in C++ known as the *iostream* library. Input coming from the user’s terminal, referred to as standard input, is “tied” to the predefined `istream cin` (pronounced “see-in”). Output directed to the user’s terminal, referred to as standard output, is tied to the predefined `ostream cout` (pronounced “see-out”). The `<<` is an output operator which means *put to*.

■ **Task 1:** If the C++ program is saved in a file called `hello.C`. You can compile and run the program like this

---

<sup>1</sup>All the examples used are available on line at `/dept/dcs/comp3062/public/examples/lab1`.

```
CC -o hello hello.C
```

Rather than explicitly writing the newline character, one can apply the predefined iostream manipular `endl`. `endl` inserts a newline character into the output stream, the output stream flushes the output buffer.

```
#include <iostream.h>

main()
{
    cout << "The sum of 7 + 3 = ";
    cout << 7 + 3;
    cout << "\n";
}
```

The last line of the above example can be:

```
cout << endl;
```

Furthermore, successive occurrences of the output operator can be concatenated.

```
#include <iostream.h>

main()
{
    cout << "The sum of 7 + 3 = " << 7 + 3 << endl;
}
```

For readability, the concatenated output statement may span several lines.

```
#include <iostream.h>

main()
{
    cout << "The sum of 7 + 3 = "
         << 7 + 3
         << endl;
}
```

■ **Task 2:** Type in the examples above. Then compile and run them.

■ **Task 3:** What does the following program do?

```
#include <iostream.h>
```

```

main()
{
    int i;
    double x;

    cout << "\nEnter a double: ";
    cin >> x;
    cout << "\nEnter a positive integer: ";
    cin >> i;
    cout << "i * x = " << i * x << endl;
}

```

## 2 Classes and Members

A class is a user-defined type. Access to objects of a class can be restricted to a set of functions declared as a part of the class. Such functions are called *member functions*. Objects of a class are created and initialised by member functions specifically declared for that purpose. Such functions are called *constructors*. A member function can be specifically declared to “clean up” objects of class when they are destroyed. Such a function is called a *destructor*.

### 2.1 Initialisation

Five examples are shown in this section to help you understand how to use constructors.

**Example 1:** A class called `student1` is defined without a constructor. In this case, C++ will also use a default constructor to allocate the memory needed for `student1`'s objects. Until you call `set()` member function, the values of `id` and `name` are **indeterminable**.

```

#include <iostream.h>
#include <string.h>

class student1 {
    int id;
    char *name;
public:
    // no constructor specified
    void set(int sid, char *str);
    void print();
};

void
student1::set(int sid, char* str) {
    name = new char[strlen(str)+1];
    strcpy(name, str);
}

```

```
        id = sid;
    }

void
student1::print()
{
    cout << name << "(" << id << ")" << endl;
}

main()
{
    student1 s10;

    s10.print();
    s10.set(100, "Albert");
    s10.print();
}
```

- **Task 4:** Understand Example 1. Then type in, compile and run it. What does the first `s10.print()` print out? Why?

**Example 2:** In Example 1, we need to use `set()` function to initialise a student object. However, in many cases, we want to initialise an object when we create the object. In Example 2, we define a default constructor `student2()` to initialise the values. In the constructor, `Id` is initialised to `-1` which is an invalid student id. It is an improvement, but usually, people want to initialise values at the same time objects are created.

```
#include <iostream.h>
#include <string.h>

class student2 {
    int id;
    char *name;
public:
    // a default constructor without arguments
    student2();
    void set(int sid, char *str);
    void print();
};

student2::student2()
{
    id = -1;
    name = NULL;
}
```

```

}

void
student2::set(int sid, char* str) {
    name = new char[strlen(str)+1];
    strcpy(name, str);
    id = sid;
}

void
student2::print()
{
    cout << name << "(" << id << ")" << endl;
}

main()
{
    student2 s10;

    s10.print();
    s10.set(100, "Albert");
    s10.print();
}

```

- **Task 5:** Understand Example 2. Then type in, compile and run it. What is the difference between the last two examples?

**Example 3:** In Example 3, instead of using member function `set()` to initialise values, we define a constructor which takes two arguments. The first is an integer, and the second is a pointer of character. So we can initialise objects when they are created.

```

#include <iostream.h>
#include <string.h>

class student3 {
    int id;
    char *name;
public:
    student3(int, char*);
    void print();
};

student3::student3(int sid, char* str)
{
    name = new char[strlen(str)+1];
}

```

```

        strcpy(name, str);
        id = sid;
    }

void
student3::print()
{
    cout << name << "(" << id << ")" << endl;
}

main()
{
    student3 s10(100, "Albert");
    s10.print();
}

```

- **Task 6:** Understand Example 3. Then type in, compile and run it. What is the difference between the last two examples?

However, because we define a constructor with two arguments, we can no longer create a student3's object without arguments as follows.

```

main()
{
    student3 s11; // error
    // ...
}

```

This suggests that C++ only creates a default constructor which takes no arguments when no user-specified constructors are defined.

- **Task 7:** Try it! How to solve this problem?

People want to have different ways to create objects. People want to use the name as its first argument, the id as its second argument. But, sometimes, people want to create a student object with the name only, because at the time of object creation the student id is not yet known. You will receive error messages when compiling Example 3 with following modifications in main().

```

main()
{
    student3 s11; // error
    student3 s12(100, "Albert"); // ok
    student3 s13("Albert", 100); // error
}

```

```

        student3 s14("Albert");    // error
        // ...
}

```

Why? It is simply because such constructors are not defined in the class definition.

■ **Task 8:** Try to compile it!

**Example 4:** In Example 4, you can do anything that you can not do in Example 3. But you may complain about the number of constructors you need to implement.

```

#include <iostream.h>
#include <string.h>

class student4 {
    int id;
    char *name;
public:
    student4(int, char*);
    student4(char*, int);
    student4(char*);
    student4();
    void print();
};

student4::student4(int sid, char* str)
{
    name = new char[strlen(str)+1];
    strcpy(name, str);
    id = sid;
}

student4::student4(char* str, int sid)
{
    name = new char[strlen(str)+1];
    strcpy(name, str);
    id = sid;
}

student4::student4(char* str)
{
    name = new char[strlen(str)+1];
    strcpy(name, str);
    id = -1;
}

```

```

student4::student4()
{
    id = -1;
    name = NULL;
}

void
student4::print()
{
    cout << name << "(" << id << ")" << endl;
}

main()
{
    student4 s10(100, "Albert");
    student4 s11("John", 101);
    student4 s12("Jeffrey");
    student4 s13;

    s10.print();
    s11.print();
    s12.print();
    s13.print();
}

```

■ **Task 9:** Understand Example 4. Then type in, compile and run it.

**Example 5:** In general, we don't want to write many constructors. In Example 4, we need four constructors. One way of reducing the number of related functions is to use default arguments. Example 5 shows that we can do the same thing as we do for Example 4, but with a fewer constructors.

```

#include <iostream.h>
#include <string.h>

class student4 {
    int id;
    char *name;
public:
    student4(int, char*);
    student4(char* str = NULL, int sid = -1);
    void print();
};

```

```
student4::student4(int sid, char* str)
{
    name = new char[strlen(str)+1];
    strcpy(name, str);
    id = sid;
}

student4::student4(char* str, int sid)
{
    if (str != NULL){
        name = new char[strlen(str)+1];
        strcpy(name, str);
    } else name = NULL;
    id = sid;
}

void
student4::print()
{
    cout << name << "(" << id << ")" << endl;
}

main()
{
    student4 s10(100, "Albert");
    student4 s11("John", 101);
    student4 s12("Jeffrey");
    student4 s13;

    s10.print();
    s11.print();
    s12.print();
    s13.print();
}
```

- **Task 10:** Understand Example 5. Then type in, compile and run it. What is the difference between the last two examples?

## 2.2 Clean Up

More often than not, a user-defined type has a constructor to ensure proper initialisation. Many types also need the inverse operation, a destructor, to ensure proper cleanup of objects of the type. The name of the destructor for class X is `~X()`.

Before discussion of destructors, we should discuss about lifetime of objects. Unless the programmer specifies otherwise, an object is created when its definition is encountered and destroyed when its name goes out of scope. Objects with global names are created and initialised once(only) and “live” until the program terminates. Local objects defined by a declaration with the keyword `static` also “live” until the end of the program.

```
void
f()
{
    student4 s11; // initialised at each call of f()
    // ...
    // destroyed when s11 goes out of the scope of f()
}
```

Destructors server two main purposes. One is to clean up the memory allocated for the members of the objects. When an object goes out of scope, C++ is able to collect the memory allocated for that object. However, C++ is unable to collect the memory that has been allocated for the members of that object. For example, in class `student4`, the constructors allocate the memory for keeping “names”, an array of character, together with the memory allocated for `student4`’s objects (the memory for keeping an integer `id` and a character pointer `name`). It would be best to free the array of characters when destroying a `student4` object. The other is to keep consistency. For example, suppose that there is a course object called `c99` which contains a list of all students enrolled in the course. When a student object is removed from the student records, the student object must be also removed from the list. Destructors are provided for cleaning up everything when objects cease to exist.

- **Task 11:** Modify the class `student4` in Example 5 with addition of a destructor as follows. Then compile and run it.

```
class student4 {
    // ...
public:
    // ...
    ~student4();
}

student4::~~student4()
{
    cout << "byebye " << name << endl;
    delete[] name;
}
```

What is the difference between `delete` and `delete []`?

## 2.3 Operator and Friend

Classes in C++ provide a facility for specifying objects, such as `student4`. Defining operators to operate on objects sometimes allows you to provide a more conventional and convenient notation for manipulating objects than could be achieved using only the basic functional notation. As we have known, the `<<` is available for primitive objects such as `int`, `char`.

```
cout << "Hello, World!\n";
```

Can we use the output operator `<<` instead of a member function `s10.print()` as follow?

```
main()
{
    student4 s10(100, "Albert");
    student4 s11("John", 101);
    student4 s12("Jeffrey");
    student4 s13;

    cout << s10 << endl;
    cout << s11 << endl;
    cout << s12 << endl;
    cout << s13 << endl;
}
```

Yes and No. You can't do it without defining `<<` in the class. Why can't you? However, you can it as follows by adding a friend operator.

■ **Task 12:** Add one line into the definition of class `student4`, as follows.

```
class student4 {
    // ..
    friend ostream& operator<< (ostream&, student4&);
}
```

Friend gives the function of `ostream& operator<< (ostream&, student4&)` permission to access `student4`'s private data. Then implement the `ostream& operator<< (ostream&, student4&)` is given as follows.

```
ostream&
operator<<(ostream& o, student4& s)
{
    return o << s.name << "(" << s.id << " ";
}
```

Test it by using the `main()` above.

## 2.4 Inheritance and Virtual Function

Object-oriented programming supports class/subclass, or type/subtype relationships. Virtual functions are functions that may be redefined later in a derived class.

- **Task 13:** A class/subclass example is given on the next page. Understand the following example. Compile and run it. Then write simple programs to answer the following questions.

- What is a base class, and what is a derived class?
- How do you define a derived class?
- Are the following the same or different?

```
class employee { /* .. */};
class manager : public employee { /* ... */};
```

and

```
class employee { /* .. */};
class manager : private employee { /* ... */};
```

and

```
class employee { /* .. */};
class manager : employee { /* ... */};
```

- What is an initialisation list? How do you initialise employee part when a manager object is going to be created?
- What is virtual function for? If you take off the keyword of `virtual` from the definition of

```
virtual void print();
```

what will happen when you run the program?

- Add a secretary class as a derived class of `employee` without redefinition of `print()`. Add a couple of secretary objects into the list of `elist` in the program above. Test the program.
- Add a `senior_manager` as a derived class of `manager` with some new attributes and modification of the `print()` member function. Add a couple of `senior_manager` objects into the list of `elist` in the program above. Test the program.
- In employee class, a member function `attach` is defined as follows.

```
void attach(employee* e){ next = e; }
```

Is it same if we redefine it as follows?

```
void attach(employee e){ next = &e; }
```

```
#include <iostream.h>
#include <string.h>

class employee {
    char* name;
    char* department;
    employee *next;
public:
    employee(char* n, char* d);
    employee* suc(){ return next;}
    void attach(employee* e){ next = e; }
    virtual void print();
};

class manager : public employee {
    short level;
public:
    manager(char* n, char* d, int l);
    void print();
};

employee::employee(char *n, char* d)
{
    name = new char[strlen(n)+1];
    strcpy(name, n);
    department = new char[strlen(d)+1];
    strcpy(department, d);
    next = NULL;
}

void
employee::print()
{
    cout << name << '\t' << department << endl;
}

manager::manager(char *n, char *d, int l)
    : employee(n, d)
{
    level = l;
}

void
manager::print()
{
```

```
        employee::print();
        cout << '\t' << "level = " << level << endl;
    }

main()
{
    manager robin("Robin", "CS", 5), brian("Brian", "CS", 5);
    employee brad("Brad", "CS"), trev("Trevor", "CS");
    employee* elist;

    elist = &robin;
    robin.attach(&brad);
    brad.attach(&brian);
    brian.attach(&trev);

    while (elist != NULL){
        elist->print();
        elist = elist->suc();
    }
}
```