CSIS/COMP 1117B
Computer Programming

Classes and Structures

Structures

- A **struct** is a special kind of class that is made up of data members only. Members of a struct is identified by ***names***.
- Individual members are referenced by qualifying the member name with the name of the object that supplies this member:

  <struct> '.' <member>
- A struct object is very often simply referred to as a struct; usually, it is quite clear whether we mean the class or the object by considering the context in which the term is being used.

An Alternate String Implementation

- An int is added to store the length of the string:

```
struct alt_string {    // alt_string is the tag of the struct (type)
    int     length;       // length of the string
    char  s[255];       // s stores the string
};
alt_string s1, s2;    // declare 2 alt_string variables s1 and s2
// declare and initialize s
// s.length = 3
// s.s[0] = ‘H’, s.s[1] = ‘K’, s.s[2] = ‘U’
alt_string s = {3, {‘H’, ‘K’, ‘U’}};
```

Input an Alternate String

```
//  get_alt_string input up to 255 characters, stop when ‘end’ is encountered
//  return true if successful and false otherwise
bool get_alt_string(istream &in, alt_string &s, char end = ‘\n’) {
    char c;
    for (s.length = 0; s.length  < 255; ) {
        in.get(c);
        if (in.fail()) return false;            // return false on input error
        if (c == end) break;                    // reach end of string
        s.s[s.length++] = c;                    // store input into string, add 1 to length
    };
    if (c != end) skip_input(in, end);   // string too long, skip remaining input
    return true;                         // it is possible to return an empty string
}
```

- Note: the first 255 characters making up an over long string will still be returned and *without any indication to the caller*!

Optional Arguments

- end is an ***optional*** argument, if omitted, '\n' is assumed.
- The following are legitimate calls to get_alt_string:

```
istream    in;
alt_string s1, s2;
get_alt_string(cin, s1, '$');
get_alt_string(cin, s2);        // end is assumed to be '\n'
get_alt_string(in, s1, '\n');   // end set explicitly to '\n'
get_alt_string(in, s2, '%');
```

- Optional arguments ***must be the last arguments***, for example, a function with 4 parameters where the last 2 are optional, it can be called with 4, 3, or 2 actuals; in the second case, the fourth parameter will take on its default value.

Another Example - Student Record

```
struct person_name {
      string first, middle, last;
};
struct student {
    person_name  name;  // member can be another struct
    long long  uid;              // long long is the same as long long int
    person_name  advisor;
};
// declare you and initialize you.name
student you = {{“you”, “and”, “you”}};
// set last name of advisor of you to CHONG
you.advisor.last = “CHONG”;
you.uid = 3132333435;
```

Lookup Student Record

```
//  a struct can be returned from function or passed as parameter
//  lookup a student using binary search
//  user need to double check the return value to determine
//     whether or not there is a match
student lookup(student sdb[ ], int n, long long key) {
    student t = {{“you”, “and”, “you”}, 0};
    int  m, u, v;
    u = 0; v = n – 1;
    while (u <= v) {
        m = (u + v) / 2;
        if (sdb[m].uid == key) return sdb[m];
        else if (sdb[m].uid > key) v = m – 1;
        else u = m + 1;
    }
    return t;        // not found
}
```

Classes

- Structures is simply a ***structuring mechanism*** for treating a group of variables as a unit.
- A **class** is a structure with additional functionalities
  - define functions that can be invoked through objects (declared to be of this class)
  - control accesses to the data members and member functions
  - define the initialization and clean up operations

A Bank Account Example

```
class account {
public:   // available outside of account
    // public member functions, definition only
    void        deposit(double a);
    bool        withdraw(double a);
    void        print_detail();
    long long   acc_num;
    string      holder;
    double      balance;
};
```

Defining Member Functions

- The definition of a member function can appear anywhere in the program (but after the class declaration).

```
void account::deposit(double a) {
    balance += a;  // same as balance = balance + a;
    cout << “Message: [“ << a << “] deposited into A/C [“
            << acc_num << “] new balance [“ << balance << “]\n”;
};
```

- The :: is called the **scope resolution operator** which indicates that deposit is a member of the class account
- Note that members can be referred to directly (without any qualification) inside a member function definition.
- It is also possible to supply the complete definition of member functions inside the class definition.

Access Control

- To ensure proper operations, it is often necessary to prevent entities outside of an object from accessing members of an object directly; this can be achieved by making them ***private***.
- For example, the account details in the bank account example should be made private and they should be accessible only to member functions.
- Member functions can also be made private if they only serve other member functions of the class.
- Public functions may be provide to read private variables (accessor functions) or to update private variables (mutator functions).
- The set of public members (data and function) is called the **interface** of the class.

Access Control -The Bank Account Class

```
class account {
public:   // available outside of account
    // public member functions, definition only
    double          deposit(double a);
    double          withdraw(double a);
    void            print_detail();
    // accessors
    long long       get_acc_num() { return acc_num; };
    string          get_holder() { return holder; };
    double          get_balance() { return balance; };
    // mutators
    void    set_acc_num(long long n) ( acc_num = n; };
    void    set_holder(string h) { holder = h; };
    void    set_balance(double b) { balance = b; };
private:  // accessible only by member functions
    long long       acc_num;
    string          holder;
    double          balance;
};
```

Constructor

- A c**onstructor** is a special function which is called automatically when an object of the class is created.
- A constructor is typically used to initialize the member variables.
- To define a constructor:
  - must have the same name as the class
  - has no return type, not even void
  - must be a public member
- C++ requires that a constructor must be called when an object is created; if a class is defined without any constructor, C++ will supply a default empty constructor:

  <class>() { };

Constructors - The Bank Account Class

```
class account {
public:   // available outside of account
    // public member functions, definition only
    double          deposit(double a);
    double          withdraw(double a);
    void            print_detail();
    . . . // accessors and mutators omitted
private:  // accessible only by member functions
    long long       acc_num;
    string          holder;
    double          balance;
public:   // constructors must be public
    account() { };    // allows declaration without initialization
    account(long long n, string p, double b = 0.0) {
        acc_num = n; holder = p; balance = b;
    };
};
```

More on Constructors

- A class may have none, one, or more constructors and the appropriate constructor is selected base on the number and type of actual parameters supplied in the initialization.
- The followings are legitimate declarations of account objects:

```
account  a1(123456789, “a person”, 100.0);
account  a2(332211445, “another person”);
account  a3;    // needs proper initialization later
```

- The empty constructor is defined explicitly because C++ will not supply it if at least one constructor has been defined for a class (and it is needed if declaration without initialization is to be allowed).

Destructor

- The destructor is a special function called automatically when an object is destroyed, e.g., when its scope ends or when an object is deleted.
- Used to perform any clean up action, e.g., release memory allocated to the object.
- To define a destructor, use the same name as the class and preceded it by tilde (~).
- Has no return type, not even void.
- Has no parameter.
- Must be a public member.
- Cannot be called explicitly using the dot (.) operator.
- If no destructor is defined for a class, C++ will supply a default empty destructor which does nothing.

Destructor – The Bank Account Class

```
class account {
public:   // available outside of account
    // public member functions, definition only
    . . .
private:  // data members, accessible only by member functions
    . . .
public:   // constructors and destructor must be public
    account() { };    // allows declaration without initialization
    account(long long n, string p, double b = 0.0) {
        acc_num = n; holder = p; balance = b;
    };
    ~account() {
        if (balance != 0.0)
            cout << “Warning: account [“ << acc_num << “] closed with balance [“
                    << balance “]\n”;
    };
};
```

Abstract Data Types

- **Encapsulation** is the provision of mechanisms for combining a number of variables and functions into a single package, such as an object of some class.
- Furthermore, accesses to individual variables and functions are provided through a well-defined interface.
- Operations internal to the object are typically invisible from outside of the object (**information hiding**). If the implementation is modified but the interface remains unchanged, users will not be affected.
- A data type is called an **abstract data type** if users of the type do not have access to details of how the values and operations are implemented.

The Bank Account Class as ADT

- The remaining methods are:

```
bool account::withdraw(double a) {
    if (balance < a) {
        cout << “Warning: insufficient fund in A/C [“ << acc_num
             << “] for withdrawal of [“ << a << “]\n”;
        return false;
    } else {
        balance –= a;  // equivalent to balance = balance – a;
        cout << “Message: [“ << a << “] withdrawn from  A/C [“
             << acc_num << “] new balance [“ << balance << “]\n”;
        return true;
    }
};
void account::print_details() {
    cout << “Account details:\n”;
    cout << “Account number [“ << acc_num << “]\n”;
    cout << “Account holder [“ << holder << “]\n”;
    cout << “Account balance [“ << balance << “]\n”;
};
```

Using the Bank Account ADT

```
int main() {
    account  a1(123456789, “person 1”, 1000.0);
    account  a2;
    account  a3(332211556, “person 3”);

    a1.deposit(500.0);
    if ( ! a1.withdraw(2000.0) ) cout << “Oops!\n”;
    a1.print_details();
    a2.set_acc_num(132333435);
    a2.set_holder(“another person”);
    a2.set_balance(300.0);
    cout << “Second A/C [“ << a2.get_acc_num() << “] opened for [“
            << a2.get_holder() << “] with balance [“ << a2.get_balance()
            << “]\n”;
    a3.print_details();
}
```

What is Missing in the Example?

- Should not use individual variables to implement individual accounts, ***why?***
  - need to organize accounts in certain ways
    - need to lookup a customer using acc_num
    - need to be able to add new accounts and delete old accounts
- Operation to create an account
  - obtain name and balance from user
  - generate a new account number
  - *is there space for one more account?*
- Operation to close an account
  - obtain number of account to be closed
  - check that there really is such an account and *close* it

Output Account Details to a File

- One very important operation in data processing is to ***backup*** the data.
- We can **overload** the insertion operator (<<) of the fstream class to take an account object and to write out the account details in some pre-defined format to a file.
- The insertion operator will need to have access to private data members of an account object; it needs to be defined as a **friend** of the account class.

Overloading An Operator

- Operators essentially are functions that can be called in the ***infix*** form; that is, instead of being called as:

  + (a, b)

  operators are ***called*** as:

  a + b

- Operator definitions are very similar to function definitions:

  <type> operator <operator> ‘(‘ <parameter list> ‘)’
  <body>

Overloading the Insertion Operator

- The following definition is added to the public section in the definition of the account class:

```
// const ensures that the account object will not be modified even
//    though it has been passed by reference; specifically, it cannot be
//    passed as a reference parameter to another function!
friend ofstream& operator << (ofstream& out, const account &a);
```

- The definition of the new operator is given after the account class definition:

```
ofstream& operator << (ofstream& out, const account &a) {
    // all the insertion operators are the original ones for ofstream
    out << a.acc_num;
    out << '%';          // output field separator
    out << a.holder;
    out << '%';
    out << a.balance;
    out << endl;       // output end of record
}
```

Backup Account Database

- The following function is used to backup one account record:

```
void backup(char file[ ], const account& a) {
    ofstream out;
    out.open(file, ios::app);  // open backup file for append
    if (out.fail()) {
        cout << “Error: cannot open [“ << file << “] for backup.\n”;
        exit(-1);
    }
    out << a;    // the overloaded insertion operator is used here
    out.close();
    if (out.fail()) {
        cout << “Error: cannot close backup file.\n”;
        exit(-1);
    }
}
```

Rules on Overloading Operators

- At least one argument of the overloaded operator must be of a class type.
- An overloaded operator can be
  - a member function of a class; the first operand is always an object of this class
  - a friend of a class (the argument being operated on)
  - an ordinary function
- ***Most existing*** operators can be overloaded
  - operator arity cannot be changed
  - operator precedence and associativity cannot be changed
- The scope resolution operator (::), the dot operator (.), the conditional operator (?:) ***cannot*** be overloaded.
- Assignments and the operators [ ] and -> have to be handled differently from what have been described here.