MM PG College Fatehabad

Object-Oriented Programming

Programming Language Concepts

Prepared by

B.Sc. CS

3rd Year/5th Sem.

Object Oriented Programming

- Over time, data abstraction has become essential as programs became complicated.
- Benefits:
    1. Reduce conceptual load (minimum detail).
    2. Fault containment.
    3. Independent program components.
       (difficult in practice).

- Code reuse possible by extending and refining abstractions.

Object Oriented Programming

- A methodology of programming
- Four (Five ?) major principles:

  1. Data Abstraction.
  2. Encapsulation.
  3. Information Hiding.
  4. Polymorphism (dynamic binding).
  5. Inheritance. (particular case of polymorphism ?)

Will describe these using C++, because ...

The C++ language

- An object-oriented, general-purpose programming language, derived from C (C++ = C plus classes).
- C++ adds the following to C:

1. Inlining and overloading of functions.
2. Default argument values.
3. Argument pass-by-reference.
4. Free store operators **`new`** and **`delete`**, instead of **`malloc()`** and **`free()`**.
5. Support for object-oriented programming, through classes, information hiding, public interfaces, operator overloading, inheritance, and templates.

Design Objectives in C++

- Compatibility. Existing code in C can be used. Even existing, pre-compiled libraries can be linked with new C++ code.
- Efficiency. No additional cost for using C++. Overheadof function calls is eliminated where possible.
- Strict type checking. Aids debugging, allows generation of efficient code.
- C++ designed by Bjarne Stroustrup of Bell Labs (now at TAMU).
- Standardization: ANSI, ISO.

Non Object-Oriented Extensions to C

- Major improvements over C.

    1. Stream I/O.
    2. Strong typing.
    3. Parameter passing by reference.
    4. Default argument values.
    5. Inlining.

We’ve discussed some of these already.

Stream I/O in C++

- Input and output in C++ is handled by streams.
- The directive `#include <iostream.h>` declares 2 streams: `cin` and `cout`.
- `cin` is associated with standard input. Extraction: `operator>>`.
- `cout` is associated with standard output. Insertion: `operator<<`.
- In C++, input is line buffered, i.e. the user must press <RTN> before any characters are processed.

Example of Stream I/O in C++

A function that returns the sum of the numbers in the file **`Number.in`**

```
int fileSum();
{
    ifstream infile("Number.in");
    int sum = 0;
    int value;
    //read until non-integer or <eof>
    while(infile >> value)
        sum = sum + value;
    return sum;
}
```

Example of Stream I/O in C++

Example 2: A function to copy **`myfile`** into **`copy.myfile`**

```
void copyfile()
{
    ifstream source("myfile");
    ofstream destin("copy.myfile");
    char ch;
    while (source.get(ch))
        destin<<ch;
}
```

Line-by-line textfile concatenation

```
  int ch;
   // Name1, Name2, Name3 are strings
   ifstream f1 (Name1);
   ifstream f2 (Name2);
   ofstream f3 (Name3);
   while ((ch = f1.get())!=-1 )
     if (ch =='\n')
       while ((ch = f2.get())!=-1) {
         f3.put(ch);
         if (ch == '\n') break;
       }
     else f3.put(ch);
}
```

Why use I/O streams ?

- Streams are type safe -- the type of object being I/O'd is known statically by the compiler rather than via dynamically tested '%' fields.
- Streams are less error prone:
  - Difficult to make robust code using **`printf`**.
- Streams are faster: **`printf`** interprets the language of '%' specs, and chooses (at runtime) the proper low-level routine. C++ picks these routines statically based on the actual types of the arguments.

Why use I/O streams ? (cont'd)

- Streams are extensible -- the C++ I/O mechanism is extensible to new user-defined data types.

- Streams are subclassable -- ostream and istream (C++ replacements for FILE*) are real classes, and hence subclassable. Can define types that look and act like streams, yet operate on other objects. Examples:
  - A stream that writes to a memory area.
  - A stream that listens to external port.

C++ Strong Typing

- There are 6 principal situations in which C++ has stronger typing than C.

1. The empty list of formal parameters means "no arguments" in C++.
   - In C, it means "zero or more arguments", with no type checking at all. Example:

```
char * malloc();
```

C++ Strong Typing (cont’d)

2. In C, it's OK to use an undefined function; no type checking will be performed. In C++, undefined functions are not allowed.

   Example:

```
main()
f( 3.1415 );
// C++: error, f not defined
// C: OK, taken to mean int f()
```

C++ Strong Typing (cont'd)

3. A C function, declared to be value-returning, can fail to return a value. Not in C++. Example:

```
double foo() {
        /* ... */
  return;
}
main() {
  if ( foo() ) { ... }
    ...
}
// C   :  OK
// C++: error, no return value.
```

C++ Strong Typing (cont'd)

4. In C, assigning a pointer of type void* to a pointer of another type is OK. Not in C++. Example:

```
int i = 1024;
void *pv = &i;
// C++: error,
// explicit cast required.
// C   : OK.
char *pc = pv;
int len = strlen(pc);
```

C++ Strong Typing (cont'd)

5. C++ is more careful about initializing arrays: Example:

```
char A[2]="hi";
 // C++: error,
 // not enough space for '\0'
//  C  : OK, but no '\0' is stored.
```

It's best to stick with `char A[] = "hi“;`

C++ Strong Typing (cont'd)

6. Free store (heap) management. In C++, we use **`new`** and **`delete`**, instead of **`malloc`** and **`free`**.

- **`malloc()`** doesn't call constructors, and **`free`** doesn't call destructors.
- **`new`** and **`delete`** are type safe.

Object-Oriented Programming

Object-oriented programming is a programming methodology characterized by the following concepts:

1. Data Abstraction: problem solving via the formulation of abstract data types (ADT's).
2. Encapsulation: the proximity of data definitions and operation definitions.
3. Information hiding: the ability to selectively hide implementation details of a given ADT.
4. Polymorphism: the ability to manipulate different kinds of objects, with only one operation.
5. Inheritance: the ability of objects of one data type, to inherit operations and data from another data type. Embodies the "*is a*" notion: a horse is a mammal, a mammal is a vertebrate, a vertebrate is a lifeform.

O-O Principles and C++ Constructs

O-O Concept	C++ Construct(s)
Abstraction	Classes
Encapsulation	Classes
Information Hiding	Public and Private Members
Polymorphism	Operator overloading, templates, virtual functions
Inheritance	Derived Classes

O-O is a different Paradigm

- Central questions when programming.
  - Imperative Paradigm:
    - What to do next ?
  - Object-Oriented Programming
    - What does the object do ? (vs. how)
- Central activity of programming:
  - Imperative Paradigm:
    - Get the computer to do something.
  - Object-Oriented Programming
    - Get the object to do something.

C vs. C++, side-by-side

C Code:

```
#include <stdio.h>
#define STACK_SIZE 10

struct Stack {
 int items[STACK_SIZE];
 int sp;
};

typedef struct Stack Stack;

void Print(Stack *s){
 int i;
 printf("Stack:\n");
 for (i=0;i<=s->sp;i++)
  printf("%d\n",s->items[i]);
}
```

C++ Code:

```
#include <iostream.h>
const int STACK_SIZE=10;

class Stack {
private:
 int items[STACK_SIZE];
 int sp;
public:
 Stack();
 ~Stack();
 void Push(int i);
 int Pop();
 int Top() const;

 friend ostream& operator<<
  (ostream& o, const Stack& s)
   { int i; o<<"Stack:"<<endl;
     for (i=0;i<=s.sp;i++)
     o<<s.items[i]<<endl;
     return o;
   };
};
```

C vs. C++, side-by-side (cont'd)

In C++, methods can appear inside the class definition (better encapsulation)

```c
void Initialize(Stack *s){
 s->sp=-1;
}

void Push(Stack *s, int i){
 s->items[++s->sp]=i;
}

int Pop(Stack *s){
 return s->items[s->sp--];
}

int Top(struct Stack *s){
 return s->items[s->sp];
}
```

```cpp
Stack::Stack()  {sp=-1;}
Stack::~Stack()  {}

void Stack::Push(int i) {
 items[++sp]=i;
}

int Stack::Pop()  {
 return items[sp--];
}

int Stack::Top() const {
 return items[sp];
}
```

C vs. C++, side-by-side (cont'd)

In C++, no explicit referencing.

Could have overloaded <<, >> for Stacks:

```
s << 1;  s >> i;
```

```
main(){
 Stack s;
 int i;
 Initialize(&s);
 Push(&s,1);
 Push(&s,2);
 Print(&s);
 i=Pop(&s);
 Print(&s);
}
```

```
int main() {
 Stack s;
 int i;
 s.Push(1);
 s.Push(2);
 cout << s;
 i=s.Pop();
 cout << s;
}
```

Structures and Classes in C++

- Structures in C++ differ from those in C in that members can be functions.
- A special member function is the "constructor", whose name is the same as the structure. It is used to initialize the object:

```
struct buffer {
    buffer()
        {size=MAXBUF+1; front=rear=0;}
    char buf[MAXBUF+1];
    int size, front, rear;
    }
```

Structures and Classes in C++

The idea is to add some operations on objects of type `buffer`:

```
struct buffer {
    buffer()  {size=MAXBUF+1;front=rear=0;}
    char buf[MAXBUF+1];
    int size, front, rear;
    int succ(int i)  {return (i+1)%size;}
    int enter(char);
    char leave();
}
```

Structures and Classes in C++

The definition (body) of a member function can be included in the structure's declaration, or may appear later.   If so, use the name resolution operator (::)

```
int  buffer::enter(char x) {
       // body of enter }
char buffer::leave() {
      // body of leave }
```

Public and Private Members

Structures and classes are closely related in C++:

```
struct x { <member-dclns> };
```

is equivalent to

```
class x { public: <member-dclns>};
```

Difference: by default, members of a structure are public; members of a class are private. So,

```
class x { <member-dclns> };
```

is the same as

```
struct x { private: <member-dclns> };
```

Header File Partitioning

```
-------------------- buf.h -----------------

const int MAXBUF = 4;

// Declaration of class 'buffer', with two
// public functions, enter and leave, and five
// private members: vector buf, data size,
// front and rear, and an increment operation.

class buffer {

public:
    buffer () { size=MAXBUF+1; front=rear=0; }
       // default constructor
    int  enter (char);
    char leave ();

private:
    char buf[MAXBUF+1];
    int size, front, rear;
    int succ (int i)  { return (i+1)%size }
};
```

Header File Partitioning (cont'd)

```
-------------------- buf.c -----------------

#include "buf.h"

// Definitions of the two functions in
// class buffer.  We use the name resolution
// operator (::)

int buffer::enter(char x) {
    if (succ(rear) == front) return 0;
    buf[rear] = x; rear = succ(rear);
    return 1;
}

char buffer::leave () {
    if (front == rear) return '\0';
    int x = buf[front]; front = succ(front);
    return x;
}
```

Header File Partitioning (cont'd)

```
------------------ buftest.c ---------------

//                    Main Program

#include "buf.h"
#include <stdio.h>
#include <math.h>

main () {
    buffer b;
      // common mistake: buffer b();
    int ch, nextch = getchar();
    while (nextch != EOF ) {
        if ( frand() >= 0.5 &&
              ((ch=b.leave()) != '\0') )
                cout.put(ch);
        else if (b.enter(nextch) )
            nextch = cin.get();
    }
    while ((ch=b.leave()) != '\0' )
        cout.put(ch);
    return 0;
}
```