CS 101 Computer Programming and Utilization

Lecture 15

Constants, pass by constant value, aliases,
Pass by reference, Pass by constant reference,
Pointers again: dynamic allocation
Pointer dereferencing, obtaining addresses of variables,
arguments to main

Mar 8,9- 2011

Prof. R K Joshi
Computer Science and Engineering
IIT Bombay
Email: rkj@cse.iitb.ac.in

Revision

- Pure functions vs. procedures
- Classes vs. primitive types
- Objects vs. primitive values
- Each instance has its own copy of the state
- Messages to objects are member function invocations
  - Parameters go in, results come out, and the member functions can access and modify the object's state

- Files as secondary storage: data can stay (persist) even after the program terminates
- Read, write operations
  - Sequential operations: the file position is moved automatically after each operation
- Sensing end of file
- Sensing unavailability of files
- Files as stream objects: we can use operators << and >>
- Operations *open(), close(), eof(), is_open()* as member function invocations on file stream objects

Constants

- An identifier can be defined as a constant of any given type.
- A value can be assigned in the declaration statement.
- A constant value cannot be changed later

Pass by Constant Value

- A change to parameter cannot be made inside the function body

```
void f (int p) {
  p = 10;  // changes p
}
void (const int p) {
   p = 10;  // not allowed!
}
```

Pass by reference

- Unlike pass by copy in which a copy of the actual parameter is sent into a function invocation, here we don't make a copy of the actual
- All accesses inside the body refer to the actual parameter location.
  - void f (int &p) { return p*p;}
    - is just fine
  - void f (int &p) { p=10; }
    - changes to p are changes to actual, since p refers to the actual due to pass by reference.

Pass by constant reference

- Pass by reference with a restriction
- All accesses inside the body do refer to the actual parameter location itself.
- But the body is prevented from making externally visible changes to the location of the actual parameter space
  - void f (const int &p) { return p*p;}
    - is just fine
  - void f (const int &p) { p=10; }
    - cannot change p since p refers to the actual!
    - Guess the output?

Aliases

- int x;
- int &y = x;
  - y is an alias for x
    - x and y refer to the same location
- changes to x are visible through y
- changes to y are visible through x
  - just as two names *paddy* and *padmanabhan* may be used to refer to the same person, names *x* and *y* refer to the same location in this case

Pointers again

- Pointer variables hold addresses
  - We can use a pointer variable for dynamic allocation, where space to variables is allocated during execution
    - int *A;
    - A = new int [10];
  - We can also use a pointer for making one variable point to different locations at different times
    - int *A, *B, *C;
    - B = new int [10];
    - C = new int [20];
    - A = B; ...... A = C;

Pointer Dereferencing

```
int *p;   int x;
p = new int;
*p=10;
x = *p;
```

- dereferencing operator '*' is used for dereferencing a pointer
- here, p is a pointer
- *p refers to the integer value
- *p can be used as lvalue, and also as rvalue

Obtaining a pointer

- A reference (an address) of a variable can be obtained
  - use referencing operator '&'
  - int x;   // an integer
  - int *p;   // a pointer to an integer
  - p = &x;   // p points to location of x
    - this is not pass by reference! though the operator is the same
    - in pass by reference, the &operator occurs in formal parameter declarations
    - in referencing, it is applied on actual variables

Null pointers

- A special value called NULL can be used to indicate that the pointer variable does not point to any location. This value can be used in comparison as given below.

```
int *p = NULL;
char *q = NULL;
...
if (p!=NULL) cout << *p;
```

character pointers

- const char *p = “how do you do\n”;
- cout << p;
- << is defined on char*

character pointers

- const char *p = “how do you do\n”;
- cout << p;
- << is defined on char*

Arguments to main

```
int main (int argc, char *argv[ ]) {


}
```

- argc is an integer: no. of parameters
- argv is an array of char* strings
  - it has argc no. of elements, i.e.,
    - argv[0] ...to.. argv[argc-1]
    - all are char* strings

Extracting typed values from argv [ ]

i.e. converting arguments..

```
#include <cstdlib>
int main (int argc, char *argv[ ]) {
  int x = atoi (argv[1];
  float f = atof (argv[2]);
}
```