COMP 2011 Notes

by Frank Chen

===

Function

```
void func(int x, float& y, char gender = 'M', bool alive = true);
```
A function with default arguments is not an overloaded function.

Function prototype = return datatype + function name + number and datatype of parameters
Declaration = function prototype
Definition = function header + function body
No two functions can have same signature even if they have different return type.
But two functions can have the same name but different signature => function overloading.

```
|---func prototype---|
int func(char, double);
    |-func signature-|
```

Function resolution
- Exact match
- Match after type promotion
  - **char**/**bool**/**short** => **int**, **float** => **double**
- Match after standard type conversion
  - between integral types
  - between floating types
  - between integral and floating types

PBR and PBV

```
int func(const int a) {return a * a;}
int a;
cout << func(a) << endl; // OK, int ==> const int

int func(int a ) {return a * a;}
const int a;
cout << func(a) << endl; // OK, const int ==> int

int func(int& a) {return a * a;}
const int a;
cout << func(a) << endl; // Error, const int =/=> int&

int func(const int& a) {return a * a;}
const int a;
cout << func(a) << endl; // OK, const int ==> const int&

int func(const int& a) {a++; return a * a;} // Error, a can't be modified

int func(int a) {a++; return a * a;}
const int a;
cout << func(a) << endl; // OK
```

Structure

```
struct Student{
  int height;
  char gender;
  char name[20];
}; <- SEMICOLON NEEDED!
```

```
Student frank = {180, 'M', "Frank"}; // OK
Student robert;
robert = {180, 'M', "Robert"}; // Error (theoretically)
robert = frank; // OK
robert.height = frank.height; // OK
robert.name = frank.name // Error
```

Struct + Array

```
Student bn[3] = { {180, 'M', "Frank"},
                  {180, 'M', "Robert"},
                  {180, 'M', "Flandia"} };
```

Pointer

```
lvalue and rvalue
lvalue: address
rvalue: value
x++: returns rvalue
++x: returns lvalue

Syntax:
int* pointer = &variable;

           *            *           *
pppointer --> ppointer --> pointer --> value
pppointer <-- ppointer <-- pointer <-- value
           &            &           &

*&var == var == &*var // in terms of value

int* pointer = &variable

int* const pointer = &variable; // const pointer
                                // variable and *pointer can be changed.
                                // pointer can not.
variable = 10; // OK
*pointer = 5;  // OK
pointer = &another_variable // Error

const int* pointer = &variable; // pointer to const object
                                // variable and pointer can be changed.
                                // *pointer can not.
variable = 10; // OK
*pointer = 5;  // Error
pointer = &another_variable // OK

const int* const pointer = &variable; // const pointer to const object
                                // variable can be changed.
                                // *pointer and pointer can not.
variable = 10; // OK
*pointer = 5;  // Error
pointer = &another_variable // Error
```

Pointer + Struct

```
Struct Student {
  double height;
  char gender;
  char[20] name;
}; <- SEMICOLON NEEDED!

Student frank;
Student* pfrank = &frank;
```

```
pfrank->height == (*pfrank).height == frank.height
```

Dynamic Allocation

```
int* pointer = new int;
// No other way to access this new memory except using pointer
delete pointer;
pointer = nullptr;  // Avoid dangling pointer
```

Linked List

```
struct node {
  int index;
  int data;
  node *next;
};

const int LENGTH = 8;
int source_data[LENGTH] = {3,1,4,1,5,9,2,6};
```

1.Creation

```
node* create_node_list(int source_data[], const int LENGTH) {
  node* head = new node;
  node *p = head;
  for (int i = 0; i < LENGTH; i++) {
    p->index = i;
    p->data = source_data[i];
    if(i < LENGTH - 1) {
      p->next = new node;
      p = p->next;
    }
    else {
      p->next = nullptr;
    }
  }
  return head;
}

node* create_single_node(int source_data) {
  node* head = new node;
  head->data = source_data;
  head->next = nullptr;
  return head;
}
```

2.Length Measurement, Printing, Searching

```
int length_of_list(const node* head)
{
  int length = 0;
  for (const node* p = head; p != nullptr; p = p->next) {
    length++;
  }
  return length;
}

void print(const node* head)
{
  for (const ll_cnode* p = head; p != nullptr; p = p->next) {
    cout << p->data;
    cout << endl;
  }
}
```

```
node* search(node* head, int target) {
  for (node* p = head; p != nullptr; p = p->next) {
    if (p->data == target) {
      return p;
    }
  }
  return nullptr;
}
```

3.Insertion

```
void insert(node*& head, int source_data, int position) {
  node* new_node = create_single_node(source_data);
  if(position == 0 || head == nullptr) {
    new_node->next = head;
    head = new_cnode;
    return;
  }
  else {
    node* p = head;
    for(int i = 0; i < position-1 && p->next != nullptr; p = p->next, i ++);
    new_node->next = p->next;
    p->next = new_node;
  }
}
```

4.Deletion

```
void delete(node*& head, int target)
{
  node* prev = nullptr;
  node* current = head;
  while (current != nullptr && current->data != target) {
    prev = current;
    current = current->next;
  }
  if (current != nullptr)  // Data is found
  {
    if (current == head) { // Special case: delete the first item
      head = head->next;
    }
    else {
      prev->next = current->next;
    }
    delete current;
  }
}
```

5.Delete All

```
void delete_all(node*& head)
{
  if (head == nullptr)
    return;
  delete_all(head->next);
  delete head;
  head = nullptr;
}
```

Other variants of linked list
circular linked list, doubly linked list, binary tree

Pointer + Array

```
int x;
int N;
int* p = &x;
p + N == &x + sizeof(int)*N

int array[5] = {1,2,3,4,5}
int* p = array;
array[0] == *array == *p == 1
&array[0] == array == p == 0x...
array[1] == *(array + 1) == *(p + 1) == 2

unsigned size;
int* pointer = new int[size];
delete[] pointer;
pointer = nullptr;


const char* word = "Hello"; // OK, only for string
char* word = "Hello"; // Maybe error, depends on compiler
const int* x = {1, 2, 3}; // WRONG!
```

Dynamic 2D Array

```
int** create_matrix(int num_rows, int num_columns) {
  int** x = new int* [num_rows];
  for (int j = 0; j < num_rows; ++j)
    x[j] = new int [num_columns];
  return x;
}

void print_matrix(const int* const* x, int num_rows, int num_columns) {
  for (int j = 0; j < num_rows; ++j)
  {
    for (int k = 0; k < num_columns; ++k)
      cout << x[j][k] << '\t';
    cout << endl;
  }
}

void delete_matrix(const int* const* x, int num_rows, int num_columns) {
  for (int j = 0; j < num_rows; j++)
    delete [] x[j];
  delete [] x;
}
```

Class

```
#include <iostream>
#include <cstring>
#include "header.h"

class Student {

  private:
    double height;
    char gender;
    char* name;

  public:

// Constructor

  Student() {height = 0; gender = '?'; name = nullptr}
                                // Name must be identical to the class name.
  Student(double h, char g, char* n) {
```

```
    height = h;
    gender = g;
    name = new char[strlen(n)+1];
    strcpy(name, n);
  }

// Accesssor

  double get_height() {return height;}
  void print_student() {
    cout << name << ": " << height << "cm, " << gender << endl;
  }

  void get_gender();

// Mutator

  void change_height(double h){height = h;}

// Destructor

  ~student(){delete[] name;}

}; <- SEMICOLON NEEDED!


int main() {
  Student frank(183, 'M', "Frank");
  frank.change_height(182);
  frank.print_student();
  return 0;
}

// "header.h"
char student::get_gender(){return gender;}
```

Stack

```
class int_stack
{
  private:
    int data[BUFFER_SIZE];  // Use an array to store data
    int top_index;          // Starts from 0; -1 when empty

  public:
    // CONSTRUCTOR
    int_stack();            // Default constructor

    // ACCESSOR
    bool empty() const;     // Check if the stack is empty
    bool full() const;      // Check if the stack is full
    int size() const;       // Give the number of data currently stored
    int top() const;        // Retrieve the value of the top item

    // MUTATOR
    void push(int);         // Add a new item to the top of the stack
    void pop();             // Remove the top item from the stack
};
```

Queue

```
class int_queue // Circular queue
{
  private:
```

```
    int data[BUFFER_SIZE]; // Use an array to store data
    int num_items;         // Number of items on the queue
    int first;             // Index of the first item; start from 0

  public:
    // CONSTRUCTOR
    int_queue();           // Default constructor

    // ACCESSOR
    bool empty() const;    // Check if the queue is empty
    bool full() const;     // Check if the queue is full
    int size() const;      // Give the number of data currently stored
    int front() const;     // Retrieve the value of the front item
    // MUTATOR
    void enqueue(int);     // Add a new item to the back of the queue
    void dequeue();        // Remove the front item from the queue
};
```

Miscellaneous

```
cin.getline(char s[], int max-num-char, char terminator);
```

Identifier name: {0-9, a-z, A-Z, _} only

```
sizeof(short) <= sizeof(int) <= sizeof(long) <= sizeof(long long)
sizeof(char) == sizeof(bool) == 1
sizeof(short) == 2
sizeof(int) == sizeof(float) == 4
sizeof(long) == sizeof(double) == 8
```

Each integral type has signed/unsigned version

Coercion

```
int + double => double + double = double

int x; double y = static_cast<double>(x);
```

Switch and Enum

```
int number;
switch(number) {
  case CONSTANT_1:
    cout << "Statement_1" << endl;
    break;  // optional
  case CONSTANT_2:
    cout << "Statement_2" << endl;
    break;
  ...
  case CONSTANT_N:
    cout << "Statement_n" << endl;
    break;
  default:
    cout << "Default Statement" << endl;
}
```

Jumps to the corresponding **case**, executes codes successively until meets **break**

```
enum Month {JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC};

int i = 0
do {
  cout << "YES" << endl;
  i++;
} while(i < 3);
```