University of Hertfordshire

7COM1025
Programming for Software Engineers
Lecture 17

Dr. Renato C. de Amorim

THE ONE HUNDRED UNDER FIFTY

Function pointer

Function pointer is a variable that stores the address of a function.

```
#include <iostream>
using namespace std;
void my_func(int x){
for (int i = 0; i<x; i++)
    cout<<i<<endl;
}
int main(){
    void (*p_f) (int);
    p_f = &my_func; //the & is optional
    p_f(2);
    (*p_f)(2);
    return 0;
}
```

Function pointer (cont)

- Since you can have pointers to functions, you can also pass a function as a parameter to another function.
- If you are writing a sorting function, you will write a finite set of comparison types (ascending, descending, etc)
- How about instead, letting the user choose how the data will be sorted by passing a function

Callback (aka 'listener') functions are also a very good example:

```
void create_button( int x, int y, const char *text, function callback_func)
```

Virtual functions are implemented behind the scenes using function pointers.

Function pointer (cont 2)

```
#include <iostream>
using namespace std;
int add(int first, int second) { return first + second; }
int subtract(int first, int second){ return first - second;}
int operation(int first, int second, int (*functocall)(int, int))
{return (*functocall)(first, second); }
int main() {
    int  a, b;
    int  (*plus)(int, int) = add;
    int (*minus)(int, int) = subtract;
    a = operation(7, 5, plus);
    b = operation(20, a, minus);
    cout << "a = " << a << " and b = " << b << endl;
    return 0;
}
```

PROBLEM 17.1

Extend the previous program so that it accepts all four basic mathematical operations.

MAP

```
#include <iostream>
#include <string>
#include <map>
using namespace std;
int main (){
    int tmp_int;
    string tmp_str;
    map<string, unsigned int> my_map;
    for (int i = 0;i<3;i++){
        cout<<"Enter name"<<endl;
        getline(cin,tmp_str);
        cout<<"Enter age"<<endl;
        cin>>tmp_int;
        cin.ignore();
        my_map.insert(pair<string, unsigned int>(tmp_str,tmp_int));
    }
    cout<<"Name:"<<endl;
    getline(cin,tmp_str);
    cout<<tmp_str<<"'s age is: "<<my_map[tmp_str]<<endl;
    cout << "All Elements"<<endl;
    for (map<string, unsigned int>::iterator it=my_map.begin(); it!=my_map.end(); it++)
        cout << it->first << " => " << it->second <<endl;
    return 0;
}
```

-It's an associative array.

- Allows mapping from one data item (the key) to another (the value).
- The key and the value may be of different data types.
- Allows only one instance of a key.
- Efficient if accessing the element by its key.
- You can iterate both directions.
- Internally elements of a map are sorted by its key.

QUEUE

```
#include <iostream>
#include <list>
#include <queue>
using namespace std;
int main (){
    queue<int, list<int> > my_queue;
//list is an example, it could be another
//container
    for (int i = 0; i<5;i++){
        int tmp_int;
        cin>>tmp_int;
        my_queue.push(tmp_int);
        }
    cout<<"Elements: "<<endl;
    while(!my_queue.empty()){
        cout<<my_queue.front()<<endl;
        my_queue.pop();
    }
    return 0;
}
```

- FIFO in terms of push, pop, front and back.
- Elements are pushed into the back, but popped from its front.
- You can choose (or even design) the underlying container! (its a container adaptor)
- No iterators.

PROBLEM 17.2

Create a class create a class student capable of holding an id number, name, age and address.
Create also a class Students which holds as many students as the user wants and has a method capable of returning an object of student given an id number.

LIST (STD CONTAINER)

```
#include <iostream>
#include <list>
using namespace std;
int main (){
    list<int> my_list;
    int index;
    for (int i = 0; i<5;i++){
        int tmp_int;
        cin>>tmp_int;
        my_list.push_back(tmp_int);
        }
    for (list<int>::iterator it = my_list.begin(); it != my_list.end(); it++)
        cout<<*it<<endl;
    cout<<"Index of element to delete (base 0): ";
    cin>>index;
    list<int>::iterator it = my_list.begin();
    advance(it,index);
    my_list.erase(it);
    for (list<int>::iterator it = my_list.begin(); it != my_list.end(); it++)
        cout<<*it<<endl;
    return 0;
}
```

- Double linked list (elements are not stored in continuous memory).
- Slow look up and access, but once a position has been found, quick insertion and deletion (opposite from a vector).
- Good for things like sorting (or anything that may require moving data within the list).
- No direct access to elements by their position. Must iterate from a known position.
- It allows iteration in both directions

STACK

```
#include <iostream>
#include <vector>
#include <stack>
using namespace std;
int main (){
    stack<int, vector<int> > my_stack;
//vector is just an example, it could be
another //container.
    for (int i = 0; i<5;i++) my_stack.push(i);
    while(!my_stack.empty()){
        cout<<my_stack.top()<<endl;
        my_stack.pop();
    }
  return 0;
}
```

- LIFO in terms of push/pop/top.
- Elements are inserted and extracted from only one end of the container.
- It's a container adaptor.
- No iterators.

PROBLEM 17.3

Remember your sorting algorithm?
Implement it using a list.