Week 12 Lecture 1

NWEN 241
Systems Programming

Jyoti Sahni

jyoti.sahni@ecs.vuw.ac.nz

Contents

- Information on Mid-term test – II and Final Exam
- Associative containers
- Revision
- Introduction to Smart Pointers (Not part of syllabus for test / final exam)

Mid-term Test - II
June 2, 2022 (Thursday) at 06:10 p.m. (45 mins.)

- CLOSED BOOK
- Syllabus – Week 7 to week 12 (except smart pointers)
- Format: Multi-choice and short answer questions
- Permitted materials:
  - Only silent non-programmable calculators or silent programmable calculators with their memories cleared are permitted in this examination.
  - No electronic dictionaries are allowed.
  - Paper foreign to English language dictionaries are allowed.

Final Test
June 17, 2022 (Friday) at 09:30 – 11:30 a.m. (2 hrs)

- Online over Blackboard
  - Syllabus – Week 1 to week 12 (except Low level programming and smart pointers)
  - Multi-choice, short answer questions
- Single attempt (Multiple attempts not allowed)
  - Find a place with stable internet connection to minimize chances of any problems.
- You will be able to go back to the questions you have attempted to make any revisions.
- In case of any problems / queries during the test please contact Alvin Valera (@Alvin.valera@ecs.vuw.ac.nz).

Associative container

- Associative containers implement sorted data structures that can be quickly searched.

- `Set`: collection of unique keys, sorted by keys

- `Map`: collection of key-value pairs, sorted by keys, keys are unique

- `Multiset`:collection of keys, sorted by keys

- `Multimap`:collection of key-value pairs, sorted by keys

Set

- Defined in <set>

The type of the elements.

```
template< class k,
class Compare = std::less<K>,class Allocator = std::allocator<K>> class set;
```

Object for performing comparison to sort (Default less than operator)

Is an handle to a heap. It used to acquire/release memory and to construct/destroy the elements in that memory.

```
set <int, greater <int> > marks;

set <int> v;
```

Set Iterators

- Bi-directional iterator

`iterator`	`Iterator used to iterate through a set.`
`const_iterator`	`Const iterator used to iterate through a set.`
`reverse_iterator`	`Iterator used to iterate backwards through a set.`
`const_reverse_iterator`	`Const iterator used to iterate backwards through a set.`

Basic Set Operations

Operation	Description
Iterators	begin(), end(), cbegin(), cend(), rbegin(), rend(), crbegin(), crend() {c – constant, r –reverse (end to beg)}
Lookup	count(key), find(key), equal_range(key), lower_bound(key), upper_bound(key) {count(key)- number of elements with a specific key. Value could be 0 or 1. find(key) –find element with a specific key equal_range(key) – range of elements matching the key Lower_bound(key) / upper_bound(key) - iterator to the first element not less than / greater than the given key
Capacity	empty(), size(), max_size()
Modifiers	clear(), insert(value /range), emplace(args), erase(pos), swap(set-to_swap_with)

```
#include <iostream>
#include <set>
#include<algorithm>
using namespace std;

void disp(set<int> a){
    for(auto i:a){
        cout<<i<<" ";
    }
}
int main(){
    set<int>even ={2,4,8,12};
    set<int>odd ={3,5,7};

    even.insert(6);

    cout<<"\nEven Set: ";
    disp(even);

    cout<<"\nOdd Set: ";
    disp(odd);
```

```
     int x =12;

    if(even.find(x)!=even.end())
        cout<<"\n"<<x<<" Found  ";
    else
        cout<<"\n"<<x<<" Not found";

    cout<<"\nAll elements above 4 are: ";
    set<int>abvkey;
    //insert a range of values
    abvkey.insert(++even.lower_bound(4),even.end());

    disp(abvkey);

}
```

```
Even Set: 2 4 6 8 12
Odd Set: 3 5 7
12 Found
All elements above 4 are: 6 8 12
```

Map

- Defined in <map>

```
template<
    class k,
    class T,
    class Compare = std::less<K>,
    class Allocator = std::allocator<std::pair<const Key, T> >
> class map;
```

Map is a sorted associative container that contains **key-value pairs** with unique keys.

Keys are sorted by using **Compare**. (default less than operator)

Map Iterators

- Bi-directional iterator

`iterator`	`Iterator used to iterate through a map.`
`const_iterator`	`Const iterator used to iterate through a map.`
`reverse_iterator`	`Iterator used to iterate backwards through a map.`
`const_reverse_iterator`	`Const iterator used to iterate backwards through a map.`

Basic Map Operations

Operation	Description
Iterators	begin(), end(), cbegin(), cend(), rbegin(), rend(), crbegin(), crend() {c – constant, r –reverse (end to beg)}
Element Access	at(key), operator[key] {at() – bound checks made, [] – no bound checks made
Capacity	empty(), size(), max_size()
Modifiers	clear(), insert(value / range), emplace(args), erase(pos), swap(map_to_swap_with)
Look-up	count(), find(key), equal_range(key), lower_bound(key), upper_bound(key)

```
#include<iostream>
#include<map>
using namespace std;

    int main ()
{
  map<int, string> stuMap ={{1,"Ram"},{2, "Sam"}};
  for(auto i=stuMap.begin(); i!=stuMap.end();i++)
  {
        cout<<i->first<<" "<<i->second<<endl;
    }

  //stuMap.insert(pair<int, string>(3,"Tom"));
  stuMap.insert({3,"Tom"});

  stuMap.insert({4,"Don"});
  stuMap[3]="Harry";
  stuMap[4]="Ana";
  cout<<"After Insertion:"<<endl;
  for(auto i=stuMap.begin(); i!=stuMap.end();i++){
        cout<<i->first<<" "<<i->second<<endl;

    }
```

```
1 Ram
2 Sam
After Insertion:
1 Ram
2 Sam
3 Harry
4 Ana
```

Revision

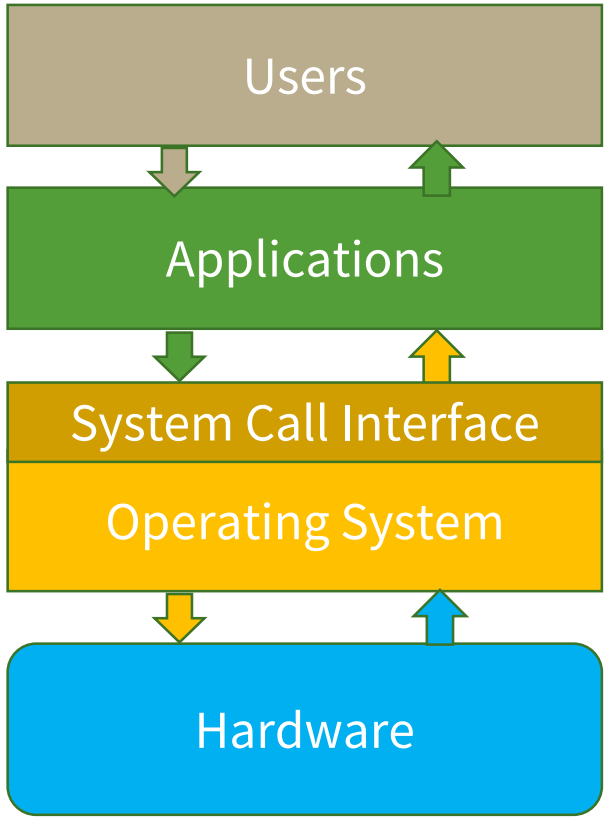
Revision

- Go to : https://app.gosoapbox.com/
- Enter Event Access code: 551-603-539

System Calls and Socket Programming

System calls

- What are system calls ?
- Why do we need them ?



Process management system calls

The following system calls are used for basic process management.

- `fork()`
- `exec()`

Defined in `unistd.h`

- `wait()`

Defined in `sys/wait.h`

- `exit()`

Defined in `stdlib.h`

What will be the output of the following program if the exec system call is sucessful?

```
#include<stdio.h>
#include<sys/wait.h>
#include<unistd.h>
#include<stdlib.h>

int main(){

    int p;
    execl("/bin/ls", "ls", NULL);
    printf("Exec successful");
   }
```

1. List of files in the current folder followed by *Exec successful* statement
2. List of files in the current folder
3. Compile time Error
4. Run time Error

What will be the output of the following program if the exec system call is sucessful?

```
#include<stdio.h>
#include<sys/wait.h>
#include<unistd.h>
#include<stdlib.h>

int main(){

    int p;
    execl("/bin/ls", "ls", NULL);
    printf("Exec successful");
   }
```

1. List of files in the current folder followed by *Exec successful* statement
2. List of files in the current folder
3. Compile time Error
4. Run time Error

What will be the output of the following program?

```
#include<stdio.h>
#include<sys/wait.h>
#include<unistd.h>
#include<stdlib.h>

int main(){

    int pid;

    fork() && fork();
    fork() && fork();

    printf("Hi\n");
   }
```

1. Hi (4 times)
2. Hi (5 times)
3. Hi (9 times)
4. Hi (16 times)

What will be the output of the following program?

```
#include<stdio.h>
#include<sys/wait.h>
#include<unistd.h>
#include<stdlib.h>

int main(){

    int pid;

    fork() && fork(); //+2 child processes
    fork() && fork();  //+ (3 x 2) child processes
    printf("Hi\n");

     }
```

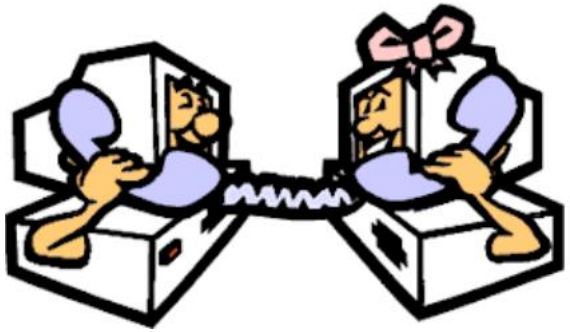
1. Hi (4 times)
2. Hi (5 times)
3. Hi (9 times)
4. Hi (16 times)

Process - Independent Vs Cooperating

- **Independent** processes: processes that don’t interact with other processes
- **Cooperating** processes: process can affect or be affected by other processes.
- In order to co-operate processes need to **communicate**
  - **Inter Process Communication**

What is socket?

- What do we need to know to allow two processes on a network to communicate?
  - **Identity of the communicating machines**
    - **IP Address**
  - **Identity of the communicating processes on these machines**
    - **Port**



- Concatenation of **IP address** and **port** defines a **socket** - A **socket** is defined as an endpoint for communication

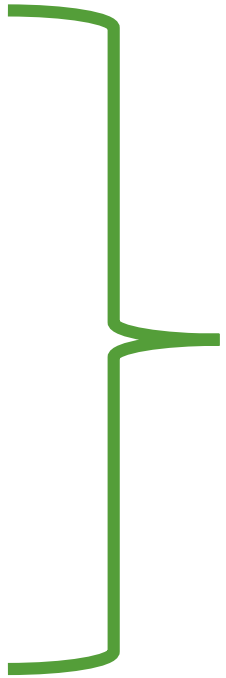
- Example: The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**

Socket types

- `SOCK_STREAM`
  - a.k.a. **TCP**
  - reliable delivery
  - in-order guaranteed
  - connection-oriented
  - bidirectional
- `SOCK_DGRAM`
  - a.k.a. **UDP**
  - unreliable delivery
  - no order guarantees
  - no notion of “connection” – app indicates dest. for each packet
  - can send or receive

System calls

- `socket()`
- `bind()`
- `listen()`
- `accept()`
- `connect()`
- `send() / sendto()`
- `recv() / recvfrom()`



Include

`sys/types.h`

`sys/socket.h`

Which type of Socket does the following statement create ?

```
fd = socket(AF_INET,SOCK_DGRAM,0) ;
```

1. UDP Socket
2. TCP Socket
3. Raw Socket
4. Reliable datagram socket

Which type of Socket does the following statement create ?

```
fd = socket(AF_INET,SOCK_DGRAM,0) ;
```

1. UDP Socket
2. TCP Socket
3. Raw Socket
4. Reliable datagram socket

Which of the following system calls is not part of a UDP connection ?

1. Socket
2. Bind
3. Connect
4. Close

Which of the following system calls is not part of a UDP connection ?

1. Socket
2. Bind
3. Connect
4. Close

Which of the following system calls is a blocking system call ?

1. Socket
2. Bind
3. Listen
4. Accept

Which of the following system calls is a blocking system call ?

1. Socket
2. Bind
3. Listen
4. Accept

Programming with C++

Primitive Data Types

Data Type	Keyword	Modifiers
Character	char	signed, unsigned
Integer	int	signed, unsigned, short, long, long long
Float	float	
Double	double	long
Boolean	bool	
Wide character representations	wchar_t	

What will be the output of the following statement?

```
cout<<float(5)/2<<" "<<float(5/2);
```

a) 2 2.5

b) 2.5 2.5

c) 2.5 2

d) 2 2

What will be the output of the following statement?

```
cout<<float(5)/2<<" "<<float(5/2);
```

a) 2 2.5

b) 2.5 2.5

c) 2.5 2

d) 2 2

Namespace

- **Namespaces** are used to organize code into logical groups and to prevent name collisions that can occur especially when your code base includes multiple libraries.
- General syntax:

```
namespace namespace_name
{
    members
}
```

- Members can be constants, variables, functions, classes, or another namespace (nested namespaces)

Example:

```
namespace myns
{
   const int N = 100;
   int count = 0;
   void printResult();
}
```

Classes

- Constructors
  - Non-parameterized
  - Parameterized
  - Copy
- Destructors
- Inline functions
- Static Members
- String class
- Friend Function
- Inheritance
  - Single, multiple inheritance
  - Virtual Functions
  - Constructors and destructors in base and derived classes (how are they defined, in what order are they invoked, explicitly invoking constructors)
  - Pure Virtual functions

How is run time polymorphism implemented ?

1. Function Overloading
2. Friend Function
3. Inheritance and Virtual Function
4. Operator Overloading

How is run time polymorphism implemented ?

1. Function Overloading
2. Friend Function
3. Inheritance and Virtual Function
4. Operator Overloading

What will be the output of the following program?

```
#include<iostream>
using namespace std;


int main(){

    int a = 10;
    int& b = a;
    b = b + 10;

    cout<<a<<" "<<b;

     }
```

a) 10 20

b) 20 20

c) Compile time error

d) Run time error

What will be the output of the following program?

```
#include<iostream>
using namespace std;


int main(){

    int a = 10;
    int& b = a;
    b = b + 10;

    cout<<a<<" "<<b;

     }
```

a) 10 20

b) 20 20

c) Compile time error

d) Run time error

Which of the following statement is true about the new and malloc?

a. The `new` is a type of operator while `malloc` is a kind of function.

b. `new` invokes a constructor, whereas `malloc` does not invoke the constructor.

c. `malloc` returns `void` pointer (requires typecasting) whereas `new` returns pointer of the desired type

- Only a
- Both a and b
- All a, b and c
- None of the above

Which of the following statement is true about the new and malloc?

a. The `new` is a type of an operator while `malloc` is a function.

b. `new` invokes a constructor, whereas `malloc` does not invoke any constructor.

c. `malloc` returns `void` pointer (requires typecasting) whereas `new` returns pointer of the desired type

- Only a
- Both a and b
- All a, b and c
- None of the above

Which of the following is a valid function declaration?

```
int sum(int x = 0, int y, int z = 0, int w = 0) ;

int sum(int x, int y = 0, int z = 0, int w = 0);

int sum(int x = 0, int y, int z, int w = 0);

int sum(int x = 0, int y = 0, int z, int w);
```

Which of the following is a valid function declaration?

```
int sum(int x = 0, int y, int z = 0, int w = 0) ;
```

```
int sum(int x, int y = 0, int z = 0, int w = 0);
```

```
int sum(int x = 0, int y, int z, int w = 0);
```

```
int sum(int x = 0, int y = 0, int z, int w);
```

Standard Template Library

Containers

- `Vector` – Dynamic, contiguous, random access
- `List` – Dynamic, non-contiguous, sequential access
- `Set` – Dynamic, sorted collection of unique elements
- `Map` – Dynamic, sorted collection of unique key value pairs

- Use cases
- Common operations - traversal, insertion, deletion, update

Which of the following is implemented through sorted data structures that can be quickly searched?

1. Array
2. Vectors
3. List
4. Set

Which of the following is implemented through sorted data structures that can be quickly searched?

1. Array
2. Vectors
3. List
4. Set

File handling

File Handling

- File Streams - `istream, ostream, fstream`
  - Newly opened streams are fully buffered by default (generally 512 bytes) or more, except streams connected to interactive devices which are line buffered
  - Stream member functions – `open(), is_open(), good(), fail(), bad(), eof()`
  - Writing the output buffer to disk is called flushing the buffer.
    - A few conditions cause the output buffer to be flushed to disk early. For example, closing the file causes any remaining data that might be hanging around in the buffer to be flushed to disk.
    - The application program can also force the output buffer to be flushed by calling `ostream::flush().`

Smart Pointers

Memory Leak

- A **memory leak** occurs when memory that was previously allocated is not properly deallocated by the programmer.

- Even though that memory is no longer in use by the program, it is still reserved, and can not be used until it is properly deallocated by the programmer.

- **Problem**: If a program has a lot of memory that hasn't been deallocated, then that could really slow down the performance of the program and can cause the program to crash.

Example

```
void fn ()
{
    // Record is a struct or class
     Record *ptr = new Record;

    int x;
    std::cout << "Enter an integer:";
    std::cin >> x;

    if (x == 0)
 // fn returns early, ptr won't be deleted!
     return;

    // do stuff with ptr here

      delete ptr;
}
```

Memory allocated for variable ptr will be leaked every time this function is called and returns early.

At heart, these kinds of issues occur because pointer variables have no inherent mechanism to clean up themselves.

Managing Memory Leaks - RAII

- **Solution**: C++ supports a programing technique - Resource Acquisition Is Initialization (`RAII`).

- `RAII` involves associating scoped objects with the acquired resources, and automatically releasing the resources once the objects are out of scope.

- C++ guarantees that the destructors of objects on the stack will be automatically executed when an object of the class goes out of scope.

- So if memory is allocated (or acquired) in a constructor, it can be deallocated in your destructor and be guaranteed that the memory will be deallocated when the class object is destroyed, regardless of whether it goes out of scope or gets explicitly deleted.

Example

```
/* Dynamic array using new and delete*/

void f(int n)
{
  int* array = new int[n];
  do_some_work(array);
  delete[] a;
}
```

Explicit call to delete required

```
/* Dynamic array using vector */
#include <vector>

void f(int n)
{
  std::vector<int> array (n);
  do_some_work(array);
}
```

Destructor called automatically as soon as the object array goes out of scope

Smart Pointer

- A smart pointer is an *object* that stores a pointer to a heap-allocated object.
- Smart pointers are typically implemented as class templates, that wrap a pointer and simulate its same behaviour, but also implement some policies to release the objects “automatically”.
- Smart pointers are always defined as stack variables and thus the invocation of their destructor is guaranteed.
- Looks and behaves like a regular C++ pointer
- With correct use of smart pointers, you no longer have to remember when to `delete` memory allocated using `new`

Smart Pointers

- C++11 standard library ships with 3 smart pointer classes:
- `std::unique_ptr`
- `std::shared_ptr`
- `std::weak_ptr`

Next lecture

- Doubt class

- Bring in your queries / doubts