NWEN 241
Systems Programming

Sue Chard

suechard@ecs.vuw.ac.nz

Content

- Review Dynamic Memory allocation C and C++
- Review Structs C and C++ C (and C++)
- Classes in C++ C++
- Destructors and Constructors C++
  - New and Delete / calloc, malloc, and free

NWEN 241 Mid-Term Test

11 April 2019, 18:00-18:45 (45 minutes)

Seating Arrangement (based on Student ID)

300035381 - 300414709: HMLT205

300415018 - 300443561: MCLT101

300443568 - 301004326: HULT323

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.

Coverage and question types:

- Test will cover Weeks 1-4 lectures & tutorials
- Will contain true or false, multiple choice, and short answer questions

Review: Dynamic memory management functions in C and C++

- calloc - allocate arrays of memory
  (n elements of “ datatype size” bytes)
- malloc - allocate a single block of memory of size bytes
- realloc - extend the amount of space allocated previously
- free - free up a piece of memory that is no longer needed by the program

Memory allocated dynamically does not go away at the end of functions, you MUST explicitly free it up

Review: Dynamic memory management functions in C++

- In addition to calloc, malloc, realloc, and free
- C++ has the new and delete keywords
  - new allocates memory on the heap
  - delete returns it to the OS
  - new and delete can be used with a wide range of data types
  - See slides in week 4 more pointers for the syntax

Memory allocated dynamically does not go away at the end of functions, you MUST explicitly free it up

Review – Common issues with Dynamic Memory

- Returning a pointer to an automatic variable
- Heap block over run (bounds errors)
- Memory leak & potential memory leak (not freeing the memory)
- Freeing non heap or unallocated memory

- Valgrind can be used to detect memory leaks and other memory management issues

What is a structure

C and C++

- In C a structure is a derived data type
  - Built using elements of other types (can be other structures)
  - An object of type struct stores data for one object
  - An array of structs stores the data for several objects

- In C++ a structure is also a derived data type
  - Built using elements of other types (can be other structures)
  - An object of type struct stores data for one object
  - An array of structs stores the data for several objects

  - In addition a C++ structure may contain member functions

Declaring Structure types

- struct declaration that only defines a type
- Variables cannot be assigned initial values

C and C++

```
struct student_info { // named struct
        char name [20];
        int student_id;
        int age;
}; // does not reserve any memory space
```

- struct declaration that defines a type and reserves storage for variables

```
struct student_info { // named struct
        char name [20];
        int student_id;
        int age;
} s, t; // reserves memory space for s and t
```

- In C++ the datatype is then used to declare variables like any other datatype

C++

```
student_info  si;
student_info   * siPtr = &si;
```

- In C the syntax requires the use of the keyword struct when declaring variables

C

```
struct student_info  si;
struct student_info   * siPtr = &si;
```

Accessing members of structures

C and C++

- Given these variable declarations

```
struct student_info  si;
struct student_info   * siPtr = &si;
```

- Dot operator .

```
si.name =  “Sam”;
si.student_id = 12345;
```

- Arrow operator -> is used in C++ when accessing a member variable of a structure object through a pointer means *dereference a pointer and then use the dot*.

```
siPtr -> name =  “Sam”;   // in C++ only
//or
(* siPtr).student_id = 12345; // in C and C++
```

C++ only Declaring Structure types with functions

C++

- struct definition that defines a type and member functions (methods).
- The function prototype appears in the structure definition
- The function is then usually defined outside the structure definition
- This code defines a print function prototype for the struct student_info
- And then defines what the function print will do
- don't have to mention what "name" and "age" are, because they automatically refer to the member variables.
- The this -> syntax may be used to be explicit
- The dot operator or arrow syntax is used to access member functions

```
struct student_info {
        char name [20];
        int student_id;
        int age;
        void print();
}sinfo;
```

```
void student_info::print() {
        cout << name << " " << this->age <<endl;
}
```

```
sinfo.print();
```

Information hiding – why?

C++

- The actual data representation used within a structure is of no concern to the structure's clients
- Protects data members from receiving invalid values
- It promotes program modifiability (if the representation of data changes, only the member functions need to change)
- The most common member access specifiers are *public* and *private*
- The *private* keyword specifies that the structure members following it are private to the structure and can only be accessed by member functions (and by *friend* functions)
- The *public* keyword specifies that the structure members following it are public to the structure and may be accessed from outside the structure

```
struct student_info {
private:
        char name [20];
        int student_id;
        int age;
public:
        void print();
};
```

C++ Classes and structures

C++

- Classes are declared in the same way as structures and share most of the same syntax
- Both define groups of variables – member variables
- Both define functions – member functions or methods

- Structure members are public by default
- Class members are private by default
- Structures are value types, classes are reference types

- Classes support inheritance structures do not

```
Definition of a class
class student_info {
private:    // not needed but included for clarity
      char name [20];
      int student_id;
      int age;
 public:
      void print(); // print member function
};


//declare an instance (object) of this class
student_info  myStudent;
```

C++ Classes / Structures and objects

C++

- Classes and structures are definitions, used to create instances of the class (or structure)
- An object is an instance of a class (or structure)
- Data and functions co-exist inside a class (or structure)
- Member functions are called without passing the data members of the class (or structure)
- There is far less chance of misusing functions by passing them the wrong data
- Changes to the internal workings of the class can be made without affecting code which uses instances of the class

C++ Constructors

C++

- A constructor is a member function which initializes an object of a class or structure, member variables cannot be assigned values in the declaration
- A constructor has:
  - the same name as the class or structure itself and has no return type
- You may declare more than one constructor for a class or structure, each one must have a different function prototype, ie different arguments / parameters

Definition of a class with constructors

```
class student_info {

      char name [20];
       int student_id;
       int age;

 public:
       void print();
       student_info();
       student_info(int);

};

student_info:: student_info(){
     student_id = 0;
}

student_info:: student_info(int id){
     student_id = id;
}

//declare an instance (object) of this class

student_info  myStudent();

//or

student_info  myStudent(12345);
```

C++ Constructors

C++

- A constructor is called automatically whenever a new instance of a class or structure is created
- You must supply the arguments to the constructor when a new instance is created
- If you do not specify a constructor, the compiler generates a default constructor for you (expects no parameters and has an empty body)

- It is also a good idea to define a copy constructor to initialize an object using another object of the same class
- If you do not specify a copy constructor the compiler will create a default one which will do a memberwise copy between the objects
- This will cause issues if a class contains pointer data members …

C++ destructors

C++

- A destructor is a member function which is called when the instance (objects) memory is about to be destroyed
- It is the clean up function
- It is usually called when the object goes out of scope
- When a class or structure contains a pointer, that the programmer dynamically allocates, it is the programmers responsibility to release the memory before the instance (object) is destroyed
- In general, if a constructor acquires resources, a destructor should release them; if a constructor establishes a relationship between entities then the destructor should terminate that relationship; and so on

```
Definition of a class

class student_info {

     char name [20];
      int student_id;
      int age;

 public:
      void print();
      student_info();
      student_info(int);
      ~student_info(); //destructor

};

student_info:: ~student info() {
      cout<< "bye";
}
```

Constructors / destructors and dynamic memory management

C++

- new calls the constructor defined for the object as it is allocating the memory
- delete calls the destructor for the object before releasing the memory

- calloc and malloc will not call the objects constructor
- realloc does not call the copy constructor
- free does not call the objects destructor – which can lead to undefined behavior if the class allocates memory internally

- When using C++ classes it is best to use new and delete unless you have good reason to use the C libraries