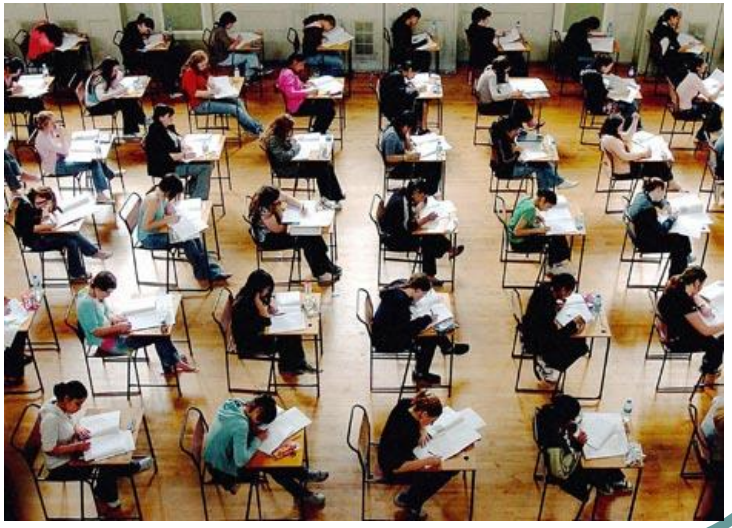
UNIVERSITY of LOUISVILLE
dare to be great

Midterm 2 Review

CECS130
Introduction to Programming Languages
Dr. Roman V. Yampolskiy

Midterm 2

- Covers C-Book chapters 10-11
- Covers C++Book chapters 1-9
- Labs 6-9
- Closed notes/books
- Worth 15% of your grade



C Chapter 10 - Dynamic allocation functions

- Dynamic allocation functions:
  - **`malloc`** – allocates space that is uninitialized
  - **`calloc`** – allocates spaces that is initialized with 0's
  - **`realloc`** – re-allocates space
  - **`free`** – de-allocates space
- Every **`malloc`**, **`calloc`**, **`realloc`** should have a matching call to **`free`**
- Otherwise, you have a *memory leak*

malloc()

```
void *malloc(size_t size);
```

- Allocate a block of `size` bytes, return a pointer to the block (`NULL` if unable to allocate block)

sizeof()

- sizeof function returns the size of a variable or data type
- Returns the number of bytes necessary to store data in memory
- sizeof can be used to find the size of any data type, variable or structure.
- Examples:

```
int temp = sizeof(int)

int myNumber = 8;
int temp = sizeof(myNumber);
```

C Chapter 11 - File Streams

•Streams are file or hardware devices such as monitor or printer

•We declare pointers to streams to get access to them

•To point to and manage a stream we use an internal data structure called FILE

•Pointers of type FILE are created like any other pointers:

FILE *pRead;

FILE *pWrite;

FILE *pAppend;

Opening Files

- fopen is used to open a file for formatted I/O and to associate a stream with that file.
- A stream is a source or destination of data.
- It may be a buffer in memory, a file or some hardware device such as a port.
- The prototype for fopen is:

  FILE *fopen(const char *filename, const char *mode);

File Opening Modes

- The valid modes are shown below:

Mode	Use
r	open for reading
w	open or create for writing. Discard any previous contents.
a	open or create for writing. Append (write after) any previous contents.

Opening Files: Example

```
#include<stdio.h>

main() {

    FILE *pRead;
    pRead = fopen("myFile.txt", "r");

}
```

goto: Example – Input Validation

```
int x = 0;

getData:
    printf("\n Please enter a positive integer: ");
    scanf("%d", &x);

if (x < 0)
     goto getData;
else printf("Thank you for entering a positive integer");
```

Error Handling: exit()

- exit() function - a way to gracefully terminate your program
- Takes a single argument
  - EXIT_SUCCESS – exit program normally
  - EXIT_FAILURE – exit program with error

Example: **exit(EXIT_SUCCESS);**

- Part of <stdlib.h> library

CPP Chapter 1 – Basic C++ Program

```
#include <iostream>

using namespace std;

int main(void) {

    cout << "Enter a character: ";
    char ch;
    cin >> ch;

    system("pause");
     return 0;
}
```

`bool` Data Type

- `bool` type
  - Has two values, `true (1)` and `false (0)`
  - Manipulate logical (Boolean) expressions
- `true` and `false` are called logical values
- `bool`, `true`, and `false` are reserved words

Function Overloading: Definitions

```
void swap (int *a, int *b)  {
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
void swap (float *c, float *d) {
    float temp;  temp = *c;  *c = *d;  *d = temp;
}
void swap (char *p, char *q) {
    char temp;  temp = *p;  *p = *q;  *q = temp;
}
```

Default Arguments: An Example

```
// Using default arguments
#include <iostream.h>

// Calculate the volume of a box
int boxVolume(int length = 1, int width = 1,
              int height = 1) {

  return (length * width * height);

}
```

Scope Resolution Operator: Example

```
int count = 0;                  // global variable

int main(void) {

    int count = 0;              // local variable
    ::count = 1;                // set global count to 1
    count = 2;                  // set local count to 2
    return 0;
}
```

Static Variables

- The syntax for declaring a static variable is:

```
static dataType identifier;
```

- The statement

```
static int x;
```

  declares x to be a static variable of the type int

- Static variables declared within a block are local to the block
- Their scope is the same as any other local identifier of that block
- Static variables have lifetimes which last until the end of the program making it possible to create functions with memory

CPP Chapter 5 - Classes

- Class: collection of a fixed number of components
- The components of a `class` are called members
- The general syntax for defining a `class`:

```
class classIdentifier
{
    classMembersList
};
```

Classes (continued)

- Three categories of class members:
  - `private`
  - `public`
  - `protected`
- By default, all members of a class are `private`
- If a member of a class is `private`
  - It cannot be accessed outside the `class`

Accessing Class Members

- Once an object is declared
  - It can access the `public` members of the class
- Syntax to access `class` members:

```
classObjectName.memberName
```

- The dot (`.`) is called the **member access operator**

Accessor and Mutator Functions

- Accessor function: member function that only accesses (does not modify) the value(s) of the member variable(s)
- Mutator function: member function that modifies the value(s) of the member variable(s)
- Constant function:
  - Member function that cannot modify member variables
  - Include reserved word `const` in function heading

Constructors

- The name of a constructor is the same as the name of the `class`.
- A constructor, even though it is a function, has no type. That is, it is neither a value-returning function nor a `void` function.
- A `class` can have more than one constructor. However, all constructors of a `class` have the same name.
- If a `class` has more than one constructor, the constructors must have different formal parameter lists.
- Constructors execute automatically when a `class` object enters its scope. Because they have no types, they cannot be called like other functions.
- Which constructor executes depends on the types of values passed to the `class` object when the `class` object is declared.

Destructors

- Destructors are functions without any type
- The name of a destructor is the character '~' followed by class name
- The name of the destructor `clockType`:

```
~clockType();
```

- A class can have only one destructor
  - It has no parameters
- The destructor is automatically executed when the class object goes out of scope

A struct Versus a Class

- By default, members of a `struct` are `public`
- By default, members of a `class` are `private`
- The member access specifier `private` can be used in a `struct` to make a member `private`
- Classes and structs have the same capabilities

CPP Chapter 5 - Inline functions

- An inline function is one in which the function code replaces the function call directly.
- Inline class member functions
  - if they are defined as part of the class definition, implicit
  - if they are defined outside of the class definition, explicit, I.e.using the keyword, *inline*.
- Inline functions should be short (preferable one-liners).
  - Why? Because the use of inline function results in duplication of the code of the function for each invocation of the inline function

Example of Inline functions

```
class CStr
{
  char *pData;
  int nLength;
    …
  public:
     …
    char *get_Data(void) {return pData; }//implicit inline function
    int getlength(void);
     …
};

inline void CStr::getlength(void) //explicit inline function
{
  return nLength;
}
    …

int main(void)
{
  char *s;
  int n;
  CStr a("Joe");
  s = a.get_Data();
  n = b.getlength();
}
```

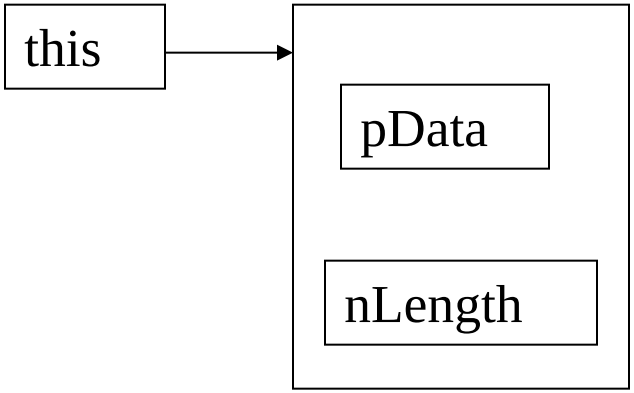
Inline functions within class declarations

Inline functions outside of class declarations

In both cases, the compiler will insert the code of the functions get_Data() and getlength() instead of generating calls to these functions

The "*this*" pointer

- Within a member function, the *this* keyword is a pointer to the current object, i.e. the object through which the function was called
- C++ passes a hidden *this* pointer whenever a member function is called
- Within a member function definition, there is an implicit use of *this* pointer for references to data members



CStr object
(*this)

Data member reference	Equivalent to
pData	this->pData
nLength	this->nLength

Chapter 6: C++ Dynamic Memory Allocation

- In C:

```
int* a = (int *) malloc(sizeof(int));
free(a);
```

- In C++:

```
int* a = new int;
delete a;
```

Chapter 7: Namespaces

•Namespaces allow to group entities like classes, objects and functions under a name.

•The global scope can be divided into subscopes

•The format of namespaces is:

namespace identifier {
entities
}

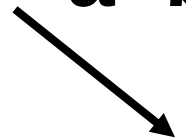
•Where identifier is any valid identifier and entities is the set of classes, objects and functions that are included within the namespace.

Implementing Operator Overloading

Defined as a member function

```
class Complex {
   ...
 public:
   ...
   Complex operator +(const Complex &op)
   {
     double real   = _real   + op._real,
                 imag = _imag + op._imag;
     return(Complex(real, imag));
   }
   ...
 };
```

c = a+b;



c = a.operator+ (b);

What is 'Friend'?

- Friend declarations introduce extra coupling between classes
  - Once an object is declared as a friend, it has access to all non-public members as if they were public
- Access is unidirectional
  - If B is designated as friend of A, B can access A's non-public members; A cannot access B's
- A friend function of a class is defined outside of that class's scope

More about 'Friend'

- The major use of friends is
  - to provide more efficient access to data members than the function call
  - to accommodate operator functions with easy access to private data members
- Friends can have access to everything, which defeats data hiding, so use them carefully
- Friends have permission to change the internal state from outside the class.
- Always use member functions instead of friends to change state

CPP Chapter 8 - Inheritance

- Inheritance is an “is-a” relationship
- For instance: “every employee is a person”
- Inheritance lets us create new classes from existing classes
- New classes are called the **derived** classes
- Existing classes are called the **base** classes
- Derived classes inherit the properties of the base classes

Inheritance

- Single inheritance: derived class has a single base class
- Multiple inheritance: derived class has more than one base class
- Can be viewed as a tree (hierarchy) where a base class is shown with its derived classes

The general syntax of a derived class is:

```
class className: memberAccessSpecifier baseClassName
{
    member list
};
```

- Where `memberAccessSpecifier` is `public`, `protected`, or `private`.
- When no `memberAccessSpecifier` is specified, it is assumed to be a `private` inheritance.

redefining vs. overloading

Overloading a function is a way to provide more than one function with the same name but with different signatures to distinguish them.

To redefine a function, the function must be defined in the derived class using the same signature and same return type as in its base class.

Pointers to a Base Class

- A pointer to a base class can store the address of a derived object

```
Base * b;
Derived d;
b = &d;
```

- Compiler assumes the stored object is of the base class
- To call a function you've overridden in a derived class need to use **virtual** functions

Virtual Functions: Syntax Example

```
class Base {
    public:
    virtual void f();
}

class Derived {
    public:
    void f() {cout<<“This is f”; };
}
```

Example of an Abstract class

```
class Figure
{
    private:
        int x_, y_;
    public:
        Figure(int x, int y) : x_(x), y_(y) {
            cout << "Figure Constructor\n";
        }
        ~Figure() {
            cout << "Figure Destructor\n";
        }

        virtual double calculateArea() = 0;
};
```

CPP Chapter 9 - Templates

- Templates: a single code body for a set of related functions (called function template) and related classes (called class template)
- The syntax for templates is:

```
template <class Type>
    declaration
```

where `Type` is the type of the data and declaration is either a function declaration or a class declaration

<typename T> preferred

- You can use either **<typename T>** or **<class T>** to specify a type parameter

- Using **<typename T>** is better because **<typename T>** is descriptive

- **<class T>** could be confused with class declaration

Default Type Arguments

C++ allows you to assign a default type for a type parameter in a class template. For example, you may assign **int** as a default type in the generic **Stack** class as follows:

```
template<typename T = int>
class Stack
{
  ...
};
```

Overloading Function Templates: Example

```
#include <iostream.h>

template <class T>
T&  min(T  &tParam1, T  &tParam2) {
   if(tParam1 < tParam2)
      return tParam1;
   else
      return tParam2;
}

template <class T>
T&  min(T  &tParam1, T  &tParam2, T  &tParam3) {
   if(min(tParam1, tParam2) < tParam3)
      return min(tParam1, tParam2) ;
   else
      return tParam3;
}
```

Test Format

- True or False
- Multiple Choice
- Trace Code
- Find Errors
- **Write Code**

True or False

- **C++ is on OOP Language**

T **F**

Multiple Choice

- Which of the following is NOT a type of member access?

A) public
B) private
C) polymorphic
D) protected

Trace Code

- What is the output produced by the following code?

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


main() {

    cout<<"Hello World!";

}
```

Find Errors

- Find all errors in the following code segment:

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


main() {

    cout < Hello World!;

}
```

Write Code

- Write a class to represent a student record
  - Name (first, last)
  - SSN
  - GPA
  - Etc.

The End!