This lecture will be a basic review of the C++ programming features. If you need more resource on Object Oriented programming and C++ check:

CS209 – Object Oriented Programming
http://homes.ieu.edu.tr/~cevrendilek/CS209_2009_10FALL.html
Why C++?

- C++ is the most used and ported programming language in the world
- C++ is a better C
- Use for C style programming
- Use for Object Oriented Programming
- Use for Generic Programming
C++ Design Goals

- Zero overhead principle
  - Don't pay for what you don't use
- No built in high level operations
  - Enables low level control and flexibility
- Efficiency
  - C++ programs run as fast as C, with good programming practices
- Improved productivity
  - Has many ready to use libraries (STL, Boost, ACE, etc.)
Who uses C++ these days anyway?

- On **Earth**
  - Microsoft Windows
  - KDE
  - Google
  - Apple (e.g., iPod user interface)
  - Mozilla Firefox, Thunderbird
  - MySQL and many more...
- On **Mars !!!**
  - Mars rover (autonomous driving system)
Compilation Model

Your code → Text file .cpp or .h → Preprocessor → Compiler → Libraries → Linker → Object code .o → Executable Code

```
g++ preprocessor compiler linker
```
Compiling example

test.cpp

```cpp
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello world" << endl;
}
```

compile:
```
g++  -O2  test.cpp -o test
```

run:
```
./test
```

See test.cpp for more...
Namespaces express logical grouping. Mostly used in complex code and libraries to avoid function name overlapping.

```cpp
namespace mySpace {
    void funcA(int a )
    {...}
    void funcB(...)
    {...}
}

namespace otherSpace{
    void funcA(int a)
    {...}
}

int x =10;
mySpace::funcA(x);
otherSpace::funcA(x);
using namespace mySpace;
funcA(x);
```
Console Output stream and redirection
You can use it instead of printf

```cpp
#include <iostream>

using namespace std;

int main(){
    cout << "Hello world" << endl;
}
```
You can print more complex stuff using cout without hassle

```
#include <iostream>

using namespace std;

int main(){
    int x = 2;
    int y = 4;
    cout << x << " times " << y << " is " << (x*y) << endl;
}
```

Notice the conversion from int to string for x,y,and product.
Console Input stream and redirection

```c++
int x;
int y;

cout << "Please enter x: " << endl;
cin >> x;
cout << "Please enter y: " << endl;
cin >> y;

cout << x << " times " << y << " is " << (x*y) << endl;
```

Notice the conversion from string to int for x and y

See input.cpp for more...
Strings

- C has character arrays
- Character arrays are limited in use
  - Cannot change size
  - Cannot add more characters to it
- In C++ use strings library instead
  - Strings library handles all low level manipulations and makes life easier!!!
Using C++ strings

#include <string>

using namespace std;

int main(){
    string s1;

    string s2 = “Hello ”;

    s1 = “World!”;

    string s3 = s2 + s1;
}

- Empty string
- Hello string
- Assigning value
- Concatenation
Compare strings

string s1 = “cs340”;
string s2 = “code”; 

if (s1!=s2) {...}

while (s1==”cs340”) {...}
Converting int to string

```cpp
#include <string>
#include <sstream>

using namespace std;

int main()
{
    int x = 5;
    std::string s;
    std::stringstream out;
    out << x;
    s = out.str();
}
```

String s1 = “Hello ”;
int x = 10;
s1 += x;  // Error!!!
# Converting string to int

```cpp
#include <string>
#include <sstream>

using namespace std;

int main(){
    int x;
    string s = "10";
    stringstream in(s);
    in >> x;
}
```
class sizeText {
    public:
    sizeText() { ... }
    sizeText(string t, int s) {...}
    void setText(string t){text=t;}
    string getText() {return text;}
    void setSize(int s)     {size=s;}
    int getSize()      {return size;}

    private:
    string text;
    int size;
};
Separating definition and implementation

- **size_text.h**
  - sizeText class definition

- **size_text.cpp**
  - sizeText class implementation

- **Your code**
class sizeText {
    public:
    sizeText();
    void setText(string t);
    string getText();
    void setSize(int s);
    int getSize();
    private:
    string text;
    int size;
};
#include "size_text.h"

sizeText::sizeText(){
    text = "Hello";
    size = 5;
}

void sizeText::setText(std::string t) {
    text = t;
}

.....
Your code

#include "size_text.h"

int main() {
    sizeText st;
    cout << st.getText() << endl;
    cout << st.getSize() << endl;
}

Compile using:

g++ test.cpp size_text.cpp -o test

or

g++ -c size_text.cpp
g++ test.cpp size_text.o -o test
Memory management with new/delete

• Use new instead of malloc
  – New can allocate space for any object
  – Like int, float, your class etc.

• Use delete instead of free
  – Free the memory allocated by new

• Be careful to free all the memory you allocate
Constructors (2)

Constructors have no return type!!!

Suitable constructor is called when the object is first created

class sizeText{
    sizeText() {
        size=5;
        text = “Hello”;
    }
    sizeText(string t, int s){
        size = s;
        text = t;
    }
    ..... 
};

sizeText myText; // Default constructor called;

sizeText myText(“hello”, 2); // parameterized constructor called
new/delete

int* x = new int(10);
....
delete x;

int* x = new int[10];
....
delete [] x;

sizeText* st = new sizeText();
....
delete st;

What happens when you delete an object?
Destructor

class test {
    public:
        test() { x = new int[10];}
    private:
        int* x;
};

What happens when you delete test?
class test {
    public:
    test() { x = new int[10];}
    ~test() { delete [] x; }
    private:
    int* x;
};
Object Creation

- C++ guarantees proper object initialization and cleanup
- Uses constructors for initialization
- Uses destructors for cleanup
Object creation (2)

- When you create a C++ object:
  1) Storage is allocated for the object
  2) Constructor is called to initialize that storage
  3) If step 2 does not happen, it is a source of serious bugs, so we can assume that constructor is always called
Avoid memory leaks

- Delete all the objects you create to avoid memory leaks.

See memtest.cpp for more...
class Text {
    public:
        string getText();
    protected:
        string text;
};

class ColoredText : public Text {
    private:
        string color;
};
Constructor/Destructor

- Default constructor of base class is called implicitly
- If a base class has constructors, they should be invoked.
- Base class is created before derived class and destroyed after derived class

See base.cpp for more...
Multiple inheritance

class A{
    protected:
    int getA();
    int a;
};

class B{
    protected:
    int getB();
    int b;
};

class C : public A, public B{
    protected:
    int getC();
    int c;
};

Can access public and protected methods/members of both A and B
Dynamic binding

Assume we have classes, and by design we want to use a single show function to print the contents of this class.

So that when we add a third C class, we don't have to change the show function.

class A {
    public:
        void print();
};

class B : public A {
    public:
        void print();
};

void show(A& item){
    item.print();
}
Dynamic binding (2)

Main problem is compiler cannot know the type of item in show function before runtime.

So warn the compiler about the dynamic type so that it can wait for the runtime for the binding.

The appropriate print method is called based on the type of the item!!!
Operator overloading changes the syntax of calling a function.

Makes the code more readable, and the object easier to use.

We already used many operators such as

cout << x << endl;
cin >> x
x+=2
Operator overloading

These operators are allowed:
**Syntax**

**Binary:** Non static member function taking 1 argument, or non member function taking two arguments

```cpp
int operator+(int) {...}
```

Notice use of operator keyword

**Unary:** Non static member function taking no arguments, or non member function taking one argument

```cpp
int operator++() {...}
```
Overload resolution

class X {
public:
    void operator+ (int);
    X (int);
};

void operator+ (X, X);
void operator+ (X, double);
void f (X a)
{
    a + 1;  // a. operator+ (1)
    1 + a;  // :: operator+ (X(1), a)
    a + 1.0; // :: operator+ (a, 1.0)
}
Pre and post overload

How do you separate
X++ from ++X

if you can only use

operator++()
Pre and post overload (2)

For ++X use:

operator++(X)
or
X::operator++()

For X++ use:

operator++(X,int)
or
X::operator++(int)
Copy constructors & Assignment operators

- The **assignment operator** is used to copy the values from one object to another already existing object.
- A **copy constructor** is a special constructor that initializes a new object from an existing object.

Cents cMark(5); // calls Cents constructor
Cents cNancy; // calls Cents default constructor
cNancy = cMark; // calls Cents assignment operator
Cents cNancy = cMark; // calls Cents copy constructor!
class Cents {
private:
    int m_nCents;
public:
    Cents(int nCents=0)
    {
        m_nCents = nCents;
    }
    // Copy constructor
    Cents(const Cents &cSource)
    {
        m_nCents = cSource.m_nCents;
    }
    Cents& operator=(const Cents &cSource);
};
Cents& Cents::operator=(const Cents &cSource)
{
    // check for self-assignment
    if (this == &cSource)
        return *this;
    // do the copy
    m_nCents = cSource.m_nCents;
    // return the existing object
    return *this;
}