01-0: **Syllabus**

- Office Hours
- Course Text
- Prerequisites
- Test Dates & Testing Policies
  - Check dates now!
- Course load

01-1: **C++ v. Java**

- We will be coding in C++ for this class
- Java is very similar to C++, with some exceptions:
  - Minor, syntactic differences
  - Memory management
  - Explicit vs. implicit pointers
  - Static compilation vs. virtual functions

01-2: **Whirlwind Tour of C++**

- C++ is a bit of a monster
- Only cover enough to get you started
- Go *extremely* quickly – holler if you want me to slow down
- Don't expect you to get it 100% right now – should get just enough that you can easily google solutions when you have questions coding
- If you already know C, this will be fairly straightforward.
- If you only know Java, it'll be a little bumpy, but you should be OK

01-3: **Java/C++ Comparison**

- Start with a simple example:
  - Java file Point.java

01-4: **Class File Management**

- C++ Classes are split into header files (.h), which describe the class data members and method prototypes, and .cpp files, which describe method bodies
- Take a look at Point.h (Simple)
  - Syntax of method declaration in .h files
  - Default Values
  - Use of public/private/protected

    - Much better protected than java!
  - m prepended to instance variable names
  - Don't forget the closing semicolon!

01-5: **.cpp Files for Classes**

- Define methods of a the class foo using the syntax:

```
<return type>
foo::<method name> (<parameters>)
{
  <method body>
}
```

01-6: **Preprocessor**

- In Java, the system finds class definitions for you – if it's in the classpath, you're golden
  - This is (partially) why Java is so strict on class file naming, and on having a single class per file
- In C++, you need to explicitly tell the compiler exactly which files you need
  - Allow some more flexibility: Can define multiple classes per file, and names don't need to match

01-7: **Preprocessor**

- Including .h files: #include
  - Not very subtle – literally including the .h file, as if it was pasted in the front of the file
  - `#include foo` is the same as pasting a copy of foo into the file at that location
  - This can lead to problems – such as multiple definitions if more than one .cpp file in a project includes the same .h file
- Preventing multiple definition
  - #define #ifdef #ifndef

01-8: **Simple class: Point**

```
#ifndef __POINT_H
#define __POINT_H
class Point
{
  public:
     Point(float X = 0,  float y = 0);
     ~Point();
     float GetX();
     float GetY();
     void SetX(float x);
     void SetY(float y);
     void Print();

  private:
     float mX;
     float mY;
};
#endif // __POINT_H
```

01-9: **Preprocessor**

- Can use the preprocessor to handle C-style constants
- Also useful for inline macros

```
#define PI 3.14159
#define min(x,y) (x < y) ? x : y
```

01-10: **Preprocessor**

- What is the output of this code? (Warning, tricky ...)

```
#include <stdio.h>
#define min(a,b) (a < b) ? a : b

int main()
{
  int i = 0;
  int j = 2;

  printf("%d\n", min(i++,j++));
  printf("%d\n", min(i++,j++));
  printf("%d\n", min(i++,j++));
}
```

01-11: **Flexibility**

- Java tries very hard to prevent you from shooting yourself in the foot.
- C++ (and C, for that matter), loads the gun for you, and helpfully points it in the correct general location of your lower body
- Example: Splitting code into .cpp and .h files:
    - You can place *all* your code in the .h file if you wish
    - Be sure to use #define and #ifdef properly!
    - Why is this a bad idea?

01-12: **Simple class: Point**

```
class Point
{
  public:
     Point(float initialX = 0,  float initialY = 0);
     ~Point();
     float GetX() { return x; }
     float GetY() { return y; }
     void SetX(float newX);
     void SetY(float newY);
     void Print();

  private:
     float x;
     float y;

};
```

01-13: **Memory Management**

- In Java, heap memory is automagically cleaned up using garbage collection
    - You can still have "garbage" in Java – how?
- In C/C++, memory needs to be explicity freed using delete
- However, there are more suble differences as well

01-14: **Stack vs. Heap**

- Java:
    - Primitives (int, float, boolean) are stored on the stack
    - Complex data structures (arrays, classes) are stored on the heap

    - Location is implicit
- C++
    - Can store anything anywhere
    - Classes declared in the Java style are stored on the stack
    - Need explicit pointers to store on the heap

01-15: **Stack vs. Heap**

```
int main()
{
    Point p1();          // I'm on the stack!
    Point *p2;
    p2 = new Point();    // I'm on the heap!
    p1.SetX(3.0);        // Use Java syntax for stack variables
    (*p2).SetY(4.0);     // Need to explicitly dereference heap
    p2->setY(4.0);       // Standard shorthand:
                         //  (*x).foo   <==>  x->foo)
}
```

01-16: **Memory Management**

- Anything you call "new" on, you need to call "delete" on to free
- Delete does not delete the pointer, it deletes what the pointer is pointing to
- The second you call delete, the data in that memory is unreliable
    - *Might* be Ok
    - *Usually* OK
    - Can lead to really nasty heisenbugs

01-17: **Memory Management**

- Arrays can be on the stack or heap as well
    - `int A1[10];`
    - `int *A2 = new int[10];`
- Arrays need to be deleted with `delete []`
    - `delete [] A2;`
- *Cannot* call delete for arrays on the stack

01-18: **Memory Management**

```
int main()
{
    Point *p = new Point();
    p = new Point();          // Memory leak!
    p.SetX(3.0);              // OK
    delete p;                 // OK
    printf("%f",p.GetX());    // Will usually work ...
                              // ... but us really, really bad
}
```

01-19: **Destructors**

- Destructor is a method that is called when a class is deleted
- Usually used to delete any memory that the class created

- Can also be used to free resources
- Similar to the java finalize method
  - Destructors are acually useful...

01-20: **Memory Management**

- Stack.h, Stack.cpp
  - What's wrong?
  - How to fix?

01-21: **Constructors**

- Problem: How do you call constructors for member variables?
  - Variables stored explicitly on the heap are not a problem – call the constructor on "new"
  - What about member variables not explicitly on the heap?

01-22: **Constructors**

```
#include "Point.h"

class Rectangle
{
 public:
  Rectangle(float x1, float y1, float x2, float y2)
  {
    // We'd like to call the constructors for mUpperLeft and
    //   mLowerRight to set up the points.  But constructors
    //   are called when variables are defined -- what to do?

  }
  Point GetUpperleft();
  Point GetLowerRight();

 private:
  Point mUpperLeft;
  Point mLowerRight;
};
```

01-23: **Constructors**

```
#include "Point.h"

class Rectangle
{
 public:
  Rectangle(float x1, float y1, float x2, float y2) :
      mUpperLeft(x1,y1), mLowerRight(x2, y2)
  {
     // We now don't need a body for this constructor
  }
  Point GetUpperleft();
  Point GetLowerRight();

 private:
  Point mUpperLeft;
  Point mLowerRight;
};
```

01-24: **Inheritance**

- Inheritance in C++ is very similar to inhertance in Java

```
class Circle : public point
{
  // Inherit all methods & data members of Point

  float mRadius;
}
```

- Inheritance can be public, private or protected – you almost always want public, that's tha Java behavoir
- Default (if you leave out modifier) is private (yes, that is odd)

01-25: **Inheritance**

- Constructors
  - When a subclass object is created, first the zero-parameter version of the superclass constructor is called, then the subclass constructor is called
  - We can explicitly call a constructor with > 0 parameters in the initialization of the subclass constructor

01-26: **Inheritance**

```
class Circle : public point
{
  Circle(float x, float y, float radius) :
     Point(x,y), mRadius(radius) { }

}
```

01-27: **Inheritance**

- See ConstructorFun.cpp for examples!

01-28: **Inheritance**

- Destructors
  - When a subclass object is destroyed (either by a delete, or by a local variable disappearing at the end of a function), first the destructor of the superclass is called, then the destructor of the subclass is called.

01-29: **Calling Superclass Methods**

- Normally, if a superclass has a method, we can call it in the subclass without any problems
- What if the *same* method is defined in both the subclass and the superclass?
  - We can call the subclass's method using the notation SuperClassName::MethodName
  - Note similarity to Namespace notation

01-30: **Calling Superclass Methods**

```
class Circle : public point
{
  Circle(float x, float y, float radius) :
     Point(x,y), mRadius(radius) { }

  void Print()
  {
     Point::Print();
     printf("Radius = %d", mRadius);
  }

  fload mRadius;

}
```

01-31: **Multiple Inheritance**

- C++ allows for multiple inheritance
  - A class can inherit from two different superclasses
  - Inherit all of the methods / instance variables from both superclasses
  - Can assign value of subclass to variable of either superclass
- Java uses interfaces to get much of the same functionality

01-32: **Multiple Inheritance**

```
class sub : public base1, public base2 {

   // Instances of class sub contain all
   //  methods and all instance variables of
   //  base1 and base2

};
```

01-33: **Includes in .h**

- It's usually considered poor form to have #includes in .h files
  - Leads to long chains of dependencies
  - Hard to see exactly what is being included
  - Include more than you need (pain for big projects)
- But rectangles require Points! What else can we do?
  - Rectangle.h actually doesn't need to know anything at all about Points, other than Point is a valid class
  - Use a forward declaration

01-34: **Constructors**

```
#ifndef RECTANGLE_H
#define RECTANGLE_H

class Point;   // Forward declaration of Point

class Rectangle
{
 public:
  Rectangle(float x1, float y1, float x2, float y2);
  Point GetUpperleft();
  Point GetLowerRight();

 private:
  Point *mUpperLeft;
  Point *mLowerRight;
};
#endif
```

01-35: **Includes in .h**

- But if we just use a forward declaration, and don't include Point.h, how do we know what can be done with a Point – what the parameters to the constructor are, and so on?
  - We don't!
  - .h files really shouldn't contain code anyway. (Simple stuff is OK, but if you start to need to have #includes in .h files, try something else!)
- Sometimes you *do* need to #include another .h file

01-36: **Includes in .h**

- Which variables require #includes, and which can be forward declared, and why?

```
class IncludeTest : public IncludeBase
{
public:

   IncludeTest() { }
   protected:

protected:

   InstanceClass1 mInstance1;
   InstanceClass2 *mInstance2;
};
```

01-37: **Includes in .h**

```
#include "IncludeBase.h"
#include "InstanceClass1.h"
class InstaceClass2;

class IncludeTest : public IncludeBase
{
public:
   IncludeTest() { }
   protected:

protected:
   InstanceClass1 mInstance1;
   InstanceClass2 *mInstance2;
};
```

01-38: **Virtual Functions**

- In Java, all methods are virtual
  - Every method call requries extra dereference
  - Always get the correct method
- In C++, methods are, by defalut, static
  - Determine at *compile time* which code to call
  - Advantages? Disadvantages?

01-39: **Virtual Functions**

```
class Base
{
public:
  void p1() { printf("p1 in Base\n");}
  virtual void p2() { printf("p2 in Base\n");}
};
class Subclass : public Base
{
public:
  void p1() { printf("p1 in Subclass\n");}
  virtual void p2() { printf("p2 in Subclass\n");}
};
int main()
{
  Base *b1 = new Base();
  Subclass *s1 = new Subclass();
  Base *b2 = s1;
  b1->p1();   b1->p2();
  b2->p1();   b2->p2();
  s1->p1();   s1->p2();
}
```

01-40: **Templates & STL**

- We'd like a generic data structure
  - Say, a generic list type
- Java method: Create a list of Objects
  - Some nasty casting needs to be done
  - Checks at runtime to make sure types match
  - (Note: modern Java has generics, similar to C++ templates)

01-41: **Templates & STL**

- We'd like a generic data structure
  - Say, a generic list type
- It would be nice to get static typing of generic list
- All checking could be done at compile time
- Templates to the rescue

01-42: **Templates & STL**

- Basic idea of templates:
  - Create a class, with some of the data types undefined
  - When we instantiate a templated class, we give the undefined types
  - Compiler replaces all of the templated type with the actual types, compiles
  - It is as if we hard-coded several versions of the class

01-43: **Templates & STL**

- TemplateStack.h / TemplateStack.cpp

01-44: **Standard Template Library**

- Group of template classes
- Handles all of the standard data structures
  - Lists, maps, sets, iterators
- Similar to the Java library – slightly more efficient

01-45: **C++ Iterators**

- C++ Iterators are similar to Java iterators
- One main difference
  - In Java, the "next" method returns the next element, and advances the iterator
  - In C++, their are separate operations for "give me the current element" and "advance the current element"
    - "Give me the next elemet" is overloaded * operator

- "Advance the current element" is overloaded ++ operator

01-46: **C++ Iterators**

```
#include <vector>

vector<int> v;

for (int i = 0; i < 10; i++)
{
   v.push_back(i);
}
for (vector<int>::iterator it = v.begin();
     it != v.end();
     it++)
{
    printf("%d", *it);
}
```

01-47: **C++ Iterators**

- Common iterator mistakes
    - Comparing iterator to NULL instead of .end()
    - Vectors of pointers

01-48: **C++ Iterators**

```
#include <vector>
#include "Point.h"

vector<Point *> points;

for (int i = 0; i < 10; i++)
{
   points.push_back(new Point(i,i*10));
}

for (vector<Point *>::iterator it = points.begin();
     it != points.end();
     it++)
{
    (*it)->Print();
}
```

01-49: **Namespaces**

- You're using a large library of code in your project
- You define a new class "foo"
- The class "foo" already in the library
    - Oops!
- What can you do?

01-50: **Namespaces**

- You're using a large library of code in your project
- You define a new class "foo"

- The class "foo" already in the library
- What can you do?
    - Create long names for each of your classes
    - Namespaces!

01-51: **Namespaces**

- Enclose your class (both .h and .cpp files) in a namespace

```
File:  foo.h
namespace <name>
{
  <standard body of .h file>
}


File:  foo.cpp
namespace <name>
{
  <standard body of .h file>
}
```

01-52: **Namespaces**

```
#ifndef POINT_H
#define POINT_H
namespace Geom {

class Point
{
 public:
  Point(float initialX = 0,  float initialY = 0);
  ~Point();
  float GetX();
  float GetY();
  void SetX(float newX);
  void SetY(float newY);
  void Print();

 private:
  float x;
  float y;

};

}
#endif
```

01-53: **Namespaces**

- Any class defined within the namespace "foo" can access any other class defined within the same namespace
- Outside the namespace, you can access a class in a different namespace using the syntax `<namespace>::<classname>`

01-54: **Namespaces**

```
namespace Geom
{

class Point;

class Rectangle
{
 public:
  Rectangle(float x1, float y1, float x2, float y2);
  Point *GetUpperleft();
  Point *GetLowerRight();

 private:
  Point *mUpperLeft;
  Point *mLowerRight;
};

}
```

01-55: **Namespaces**

```
class Geom::Point;

class Rectangle
{
 public:
  Rectangle(float x1, float y1, float x2, float y2);
  Geom::Point *GetUpperleft();
  Geom::Point *GetLowerRight();

 private:
  Geom::Point *mUpperLeft;
  Geom::Point *mLowerRight;
};
```

01-56: **Namespaces**

- All of the classes in the STL use the namespace std
- So, our code for vectors and iterators (above) won't *quite* compile, need to add std:: namespace reference

01-57: **Namespaces**

```
#include <vector>
#include <stdio.h>

int main()
{
  std::vector<int> v;

  for (int i=0; i < 10; i++)
    v.push_back(i);

  for (std::vector<int>::iterator it = v.begin();
     it != v.end();
     it++)
  {
    printf("%d", *it);
  }

  return 0;
}
```

01-58: **Using Namespaces**

- Using std:: everywhere can get a little cumbersome
- We certainly don't want to put our code in the std namespace
- using to the rescue

01-59: **Using Namespaces**

```
#include <vector>
#include <stdio.h>

using namespace std;

int main()
{
  vector<int> v;

  for (int i=0; i < 10; i++)
    v.push_back(i);

  for (vector<int>::iterator it = v.begin();
     it != v.end();
     it++)
  {
    printf("%d", *it);
  }

  return 0;
}
```

01-60: **Using Namespaces**

- It may be tempting to have:
  - using namespace Ogre

in your project code, since *everything* in Ogre is in the namespace Ogre

- I *strongly recommend* that you do *not* do this

01-61: **More Namespaces**

- Namespaces can nest

```
namespace foo {
  namespace bar {
     class Myclass { ... }
  }
}

...

foo::bar::Myclass x;
```

01-62: **Explicit Pointers**

- Sometimes hear “Java Has no pointers”
  - Of course this is completely incorrect
  - Java has no *explicit* pointers
- C++ has Explicit pointers, just like C (and implicit ones, too!)
- C++ is a superset of C: Every crazy thing you can do in C, you can do in C++

01-63: **Explicit Pointers**

```
int main()
{
    int x = 3;

    int *ptrX = &x;
    int *ptrA = new int;

    *ptrA = 4;
    *ptrX = 5;

    printf("ptrA = %d, *ptrA = %d", ptrA, *ptrA);
    printf("x = %d",x);
}
Output:
ptrA = 1048912, *ptrA = 4
x = 5
```

01-64: **Explicit Pointers**

- What happens if you run this in Java? C/C++?

```
int main()
{
  int A = 1;
  int x[5];
  int B = 2;
  x[-1] = 9;
  x[-2] = 10;
  x[5] = 11;
  x[6] = 12;
  printf("%d, %d \n", A, B); // (assuming Java had printf ...)
}
```

01-65: **Explicit Pointers**

- What happens if you run this in Java? C/C++?

```
int main()
{
  int A = 1;
  int x[5];
  int B = 2;
  x[-1] = 9;
  x[-2] = 10;
  x[5] = 11;
  x[6] = 12;
  printf("%d, %d \n", A, B); // (assuming Java had printf ...)
}
Java:  Runtime Error
C:  9, 10
```

01-66: **Explicit Pointers**

```
int main()
{
  int *x = new int[4];
  int *y = new int[4];

  for (int i = 0; i < 5; i++)
    {
      x[i] = i;
      y[i] = i + 10;
    }
   for (int i = 0; i < 4; i++)
    {
      printf("%d, %d, %d \n", i, x[i], y[i]);
    }
}
```

Output?

01-67: **Explicit Pointers**

```
int main()
{
  int *x = new int[4];
  int *y = new int[4];

  for (int i = 0; i < 5; i++)
    {
      x[i] = i;
      y[i] = i + 10;
    }
   for (int i = 0; i < 4; i++)
    {
      printf("%d, %d, %d \n", i, x[i], y[i]);
    }
}
Output
0,0,4
1,1,11
2,2,12
3,3,13
```

01-68: **Explicit Pointers**

```
int main()
{
  int *x = new int[5];
  int *y = new int[5];

  for (int i = 0; i < 6; i++)
    {
      x[i] = i;
      y[i] = i + 10;
    }
   for (int i = 0; i < 5; i++)
    {
      printf("%d, %d, %d \n", i, x[i], y[i]);
    }
}
```

Output?

01-69: **Explicit Pointers**

```
int main()
{
  int *x = new int[5];
  int *y = new int[5];
  for (int i = 0; i < 6; i++)
    {
      x[i] = i;
      y[i] = i + 10;
    }
   for (int i = 0; i < 5; i++)
    {
      printf("%d, %d, %d \n", i, x[i], y[i]);
    }
}
```

```
Output
0,0,10
1,1,11
2,2,12
3,3,13
4,4,14
```

(!) 01-70: **Why does this matter?**

- Could have a bug like the second example above – hidden!
- Change the size of one of your data structures
- Bug suddenly appears, *apparently* unrelated to the change you just made in the code

01-71: **Explicit Pointers**

- When you do non-standard access strange things happen
  - C++ doesn't protect you
  - Can be difficult to debug ...
- Game programming optimizes for speed – do some funky pointer manipulation, and raw access of data
- Need to have good debug-fu
  - Discuss some debug strategies later in the semester

01-72: **Pass by Reference**

- C++ allows you to pass a parameter by *reference*
- Actually pass a pointer to the object, instead of the object itself

01-73: **Pass by Reference**

```
void foo(int x, int &y)
{
   x++;
   y++;
}

int main()
{
   int a = 3;
   int b = 4;
   foo(a,b);
   printf("a = %d, b = %d",a,b);
}

Output:
a = 3, b = 5
```

01-74: **Pass by Reference**

```
void foo(int x, int *y)
{
   x++;
   (*y)++;
}

int main()
{
   int a = 3;
   int b = 4;
   foo(a,&b);
   printf("a = %d, b = %d",a,b);
}

Output:
a = 3, b = 5
```

01-75: **More References...**

- C++ allows references outside of parameters, too.

```
int main()
{
   int x = 3;
   int *y = &x;
   ...
   *y = 6;  // Now x == 6, too
}
```

01-76: **More References...**

- C++ allows references outside of parameters, too.

```
int main()
{
   int x = 3;
   int &y = x;
   ...
   y = 6;  // Now x == 6, too
}
```

- This allows for implicit pointers
- Handy for defining operators

01-77: **References vs. Pointers**

Pointers	References
Explicit, need *	Implicit, don't use *
Need not be initialized	Must be initialized
Can change what it points to	always points to the same thing
Can be null	Must point to something

01-78: **More References...**

- A function can return a reference
  - Just like a function returning a pointer
  - Don't need to explicitly follow the pointer, using *

01-79: **More References...**

```
#include <stdio.h>
#include <libc.h>
char &FirstChar(char *str)
{
  return str[0];
}

int main()
{
  char *message = new char[6];
  strcpy(message, "Hello");
  char &first = FirstChar(message);
  first = 'x';
  printf("%s\n",message);
}

Output:  xello
```

01-80: **More References...**

```
#include <stdio.h>
#include <libc.h>
char &FirstChar(char *str)
{
  return str[0];
}

int main()
{
  char *message = new char[6];
  strcpy(message, "Hello");
  FirstChar(message) = 'y';
  printf("%s\n",message);
}

Output:  yello
```

01-81: **More References...**

```
#include <stdio.h>
#include <libc.h>
char &FirstChar(char *str)
{
  return str[0];
}

int main()
{
  char *message = new char[6];
  strcpy(message, "Hello");
  char first = FirstChar(message);
  first = 'x';
  printf("%s\n",message);
}

Output:  hello
```

01-82: **More References...**

- What's wrong with me?

```
int &Foo(int x)
{
  return x
}
```

01-83: **More References...**

- What's wrong with me?

  ```
  int &Foo(int x)
  {
    return x
  }
  ```

- Returning a pointer to an element on the stack, that is immediately going away!

01-84: **Const Access**

- Sometimes we want to return a pointer to a large data structure
  - Copying all of the data would take too much time / memory
- *But,* we don't want the variablet to be modified...
- If we have a const pointer or reference, we cannot change what it points to

01-85: **Const Access**

- This compiles, but crashes with a bus error:

```
#include <stdio.h>

char *GetText()
{
  return "Hello There!";
}

int main()
{
  GetText()[3] = 'a';
}
```

01-86: **Const Access**

- This doesn't compile

```
#include <stdio.h>

const char *GetText()
{
  return "Hello There!";
}

int main()
{
  GetText()[3] = 'a';
}
```

01-87: **Const Access**

- Of course there are some tricky bits (there are always tricky bits ...)
- Does the const apply to the pointer, or what is being pointed to?
  - Const applies to the closest item on the left, or the item on the right if there is nothing on the left

01-88: **Const Access**

```
  int x, y, z;
  const int *xPtr = &x;
  int const *yPtr = &y;
  int *const zPtr = &z;

  xPtr  = yPtr;  // OK -- the value is const, not pointer
  zPtr  = yPtr;  // OK -- the value is const, not pointer
  *zPtr = 3;     // OK -- the pointer is const, not the value

  *xPtr = 5;     // BAD -- the value is const
  *yPtr = 5;     // BAD -- the value is const
  zptr  = xPtr;  //  BAD -- the pointer is const
```

01-89: **Const Access**

```
class Foo
{
  public:
    int x;
    void foo();
};

int main()
{
  Foo f;
  const Foo *fooPtr = &f;
  int z = fooPtr->x;  // OK?
  fooPtr.x = 3;       // OK?
  fooPtr.bar();       // OK?
}
```

01-90: **Const Access**

```
class Foo
{
  public:
    int x;
    void foo();
};

int main()
{
  Foo f;
  const Foo *fooPtr = &f;
  int z = fooPtr->x;  // OK -- only getting value
  fooPtr.x = 3;       // BAD -- setting value
  fooPtr.bar();       // BAD -- bar *might* set value
}
```

01-91: **Const Access**

```
class Foo
{
  public:
  int x;
  void foo() const;  // This const says that the method
                     // will not change any method variables
                     //  (nor will it call any non-const
                     //   methods)
};

int main()
```

```
{
  Foo f;
  const Foo *fooPtr = &f;
  int z = fooPtr->x;  // OK -- only getting value
  fooPtr.bar();       // OK -- bar is const, and OK
}
```

01-92: **Const Access**

- If a method is const, you can't change any of the instance variables

```
class Foo
{
  public:
    int x;
    void bar()
    {
      // Does nothing
    }
    void foo() const
    {
       x = 3;  // Illegal, const nethods can't change values
       bar();  // Illegal, can only call const methods
               //   (even though bar doesn't do anything)
    }

};
```

01-93: **Const Access**

- Const access is infectous
  - You can assign a non-const value to a const variable
  - Can't go back – once a variable is const, can't change it, or assign it to a non-const variable
    - (Though if you do have non-const access through another pointer, you can still change it, of course)
- Thus it's useful to denote any function that doesn't change instance variables as const

01-94: **Const Access**

- We can use const access for parameters to methods & functions, too

```
int foo(const int * p1, const Stack &S);
```

Which of the following are legal?

```
int foo(Stack S);
int bar(Stack &S);
int foobar(const Stack &S);
...

const Stack constStack = getStack();
foo(constStack);
bar(constStack);
foobar(constStack);
```

01-95: **Const Access**

- We can use const access for parameters to methods & functions, too

```
int foo(const int * p1, const Stack &S);
```

Which of the following are legal?

```
int foo(Stack S);
int bar(Stack &S);
int foobar(const Stack &S);
...

const Stack constStack = getStack();
foo(constStack);                      // Legal (why?)
bar(constStack);                      // Illegal
foobar(constStack);                   // Legal
```

01-96: **Operator Oveloading**

- Let's say you are writing a complex number class in Java
  - Want standard operations: addition, subtraction, etc
  - Write methods for each operation that you want (see code)
- It would be nice to use built-in operators

```
Complex c1 = new Complex(1,2);
Complex c2 = new Complex(3,4);
Complex c3 = c1 + c2;
```

01-97: **Operator Oveloading**

- In C++ you can overload operators
- Essentially just "syntactic sugar"
- Really handy for things like vector & matrix math
  - Ogre math libraries make heavy use of operator overloading
- See C++ Complex code example
- Aside: Why no operator overloading in Java?

01-98: **Operator Oveloading**

- Let's take a look at the + operator:

  ```
  const Complex operator+ (const Complex& c) const;
  ```

- Why pass in a const reference?
- Why is the return value const?

01-99: **Operator Oveloading**

- If the return value was not const:

  ```
  Complex operator+ (const Complex& c) const;
  ```

- We could do things like this:

  ```
  Complex c1, c2, c3;
  ...
  (c1+c2) = c3;
  ```

- The const return value prevents this

01-100: **Operator Oveloading**

- What happens when you assign one class to another (both stored on the stack)?

- Shallow copy

```
class DeepCopy
{
  public:
    DeepCopy(int initVal)
    {
      mPtr = new int;
      *mPtr = initVal;
    }
    int *mPtr;
};
```

01-101: **Operator Oveloading**

- What happens when you assign one class to another (both stored on the stack)?
- Values are copied across
    - Shallow copy
- What if we want a deep copy?
    - Overload the assignment operator

01-102: **Operator Oveloading**

```
class DeepCopy
{
   int *mPtr;

   // You'll want other methods (including destructor!)

   DeepCopy& operator= (DeepCopy const &rhs) {
      if (this != &rhs) {
         delete  mPtr;
         mPtr = new int;
         (*mptr) = (*rhs.mPtr)
      }
      return *this;
   }
};
```

01-103: **Operator Oveloading**

- Why do we need to check for self-assignment

```
class DeepCopy
{
   int *mPtr;

   // You'll want other methods (including destructor!)

   DeepCopy& operator= (DeepCopy const &rhs) {
      if (this != &rhs) {   <-- Why is this if test needed?
         delete  mPtr;
         mPtr = new int;
         (*mptr) = (*rhs.mPtr)
      }
      return *this;
   }
};
```

01-104: **Copy Constructors**

- Assignment operator (either the default, or user-created) when a value is copied into an existsing variable.
- When a new location is being created, copy constructor is used instead.

01-105: **Copy Constructors**

```
class DeepCopy
{
   int *mPtr;

   DeepCopy (const DeepCopy &rhs) {
         mPtr = new int;
         (*mptr) = (*rhs.mPtr)
   }
};
```

01-106: **Copy Constructors**

```
MyClass a,b;
a = b;           // = operator used
MyClass c = a;   // Copy Constructor used

void foo(MyClass x);

foo(a);          // Copy constructor used
```

01-107: **Copy Constructors**

- If you are using a copy constructor, you probably also want to overload assignment = (and vice-versa)
- If you have a copy constructor & overloaded assignment, you probably want a destructor (why?)
- Why does C++ have both copy constructors and overloading of =?

01-108: **Operator Overloading**
01-109: **Variable Initialization**

- C++ Does not initialize anything for you
- Value of any uninitialzed variable is whatever value happened to be on the stack at that locaation
- Compiler will often give a warning if you access an uninitialized variable
  - But don't count on compiler warnings! (Don't ignore them, either!)

01-110: **Runtime Checks**

- C++ has no bounds checking on arrays
- C++ has no null check on dereferencing pointers
  - Pointers can be uninitialized garbage
  - Pointers can point to deallocated memory

01-111: **More Arrays**

- Arrays in C++ are not first-class objects
  - Only a list of data
  - No length, etc
- Memory created with new foo[x] needs to be deleted with delete [] y

01-112: **Const fun**

```
const int *const
MyClass::foo(const int *bar) const;
```

01-113: **Const fun**

- What does this const mean?

- Is it meaningful for the interface?
- Does it do anything?

```
void
MyClass::foo(int *const bar);
```

01-114: **Const fun**

```
const Ogre::SceneManager *getSceneManager();

...

Ogre::SceneNode *node = mWorld->getSceneManager()->getSceneNode("cubenode");
Vector3 currentPos = node->getPosition();
node->setPosition(currentPos + move * time);
```