The University Of Michigan

EECS402 Lecture 11

Andrew M. Morgan

Savitch Ch 6
Misc. Class/Object Topics

Compiler Error Example

```
class StudentClass
{
  public:
    string name;
    int    id;
  public:
    StudentClass(const string &inName,
                 const int inId);
};
//Global function prototypes are here
int main()
{
  StudentClass sl;
  sl.name = "Pikachu";
  sl.id = 4827;
  printStudentInfo(sl);
  return 0;
}

StudentClass::StudentClass(
     const string &inName,
     const int inId
     )
{
  name = inName;
  id = inId;
}
```

```
void printStudentName(
     StudentClass &student
     )
{
  cout << student.name;
}

void printStudentInfo(
     const StudentClass &student
     )
{
  cout << "Name: ";
  printStudentName(student);
  cout << " Id: " << student.id << endl;
}
```

c.cpp: In function 'int main()':
c.cpp:56: no matching function for call to 'StudentClass::StudentClass()'

Recall: Once a ctor is provided, the default ctor is no longer available (unless a default ctor is provided as an overloaded ctor)

Fix One Error, Add Another

```
class StudentClass
{
  public:
    string name;
    int    id;
  public:
    StudentClass(const string &inName,
                 const int inId);
};
//Global function prototypes are here
int main()
{
  StudentClass sl("Pikachu", 4827);
  printStudentInfo(sl);
  return 0;
}

StudentClass::StudentClass(
     const string &inName,
     const int inId
     )
{
  name = inName;
  id = inId;
}
```

```
void printStudentName(
     StudentClass &student
     )
{
  cout << student.name;
}

void printStudentInfo(
     const StudentClass &student
     )
{
  cout << "Name: ";
  printStudentName(student);
  cout << " Id: " << student.id << endl;
}
```

c.cpp: In function `void printStudentInfo(const StudentClass &)':
c.cpp:50: conversion from `const StudentClass' to `StudentClass &' discards qualifiers
c.cpp:41: in passing argument 1 of `printStudentName(StudentClass &)'

Note: printStudentInfo() has "promised" that the student object passed in will not be changed, via the keyword const in the parameter list.

The object is then passed into the printStudentName() function, which does NOT make the same promise. Even though the object isn't changed, the compiler sees that it CAN change, and provides the compiler error shown.

Updated With No Compiler Errors

```
class StudentClass
{
  public:
    string name;
    int    id;
  public:
    StudentClass(const string &inName,
                 const int inId);
};
//Global function prototypes are here
int main()
{
  StudentClass sl("Pikachu", 4827);
  printStudentInfo(sl);
  return 0;
}

StudentClass::StudentClass(
     const string &inName,
     const int inId
     )
{
  name = inName;
  id = inId;
}
```

```
void printStudentName(
     const StudentClass &student
     )
{
  cout << student.name;
}

void printStudentInfo(
     const StudentClass &student
     )
{
  cout << "Name: ";
  printStudentName(student);
  cout << " Id: " << student.id << endl;
}
```

Name: Pikachu Id: 4827

New Version Using Member Function, Error

```
class StudentClass
{
  public:
    string name;
    int    id;

  public:
    StudentClass(const string &inName,
                 const int inId);
    string getName();
    int getId();
};
//Global function prototypes are here
int main()
{
  StudentClass sl("Pikachu", 4827);
  printStudentInfo(sl);
  return 0;
}
StudentClass::StudentClass(
     const string &inName,
     const int inId
     )
{
  name = inName;
  id = inId;
}
```

```
int StudentClass::getId(
     )
{
  return id;
}

string StudentClass::getName(
     )
{
  return name;
}

void printStudentInfo(
     const StudentClass &student
     )
{
  cout << "Name: ";
  cout << student.getName();
  cout << " Id: " << student.id << endl;
}
```

c.cpp: In function `void printStudentInfo(const StudentClass &)':
c.cpp:50: passing `const StudentClass' as `this' argument of `class string StudentClass::getName()' discards qualifiers

Same problem as earlier, but now it's a member function that isn't making the promise not to change the object! printStudentInfo() promises not to change the student object passed in, but then the student object is used to call the member function getName(), which may change the data members of the object.

Member Function Version – Error Fixed

```
class StudentClass
{
  public:
    string name;
    int    id;

  public:
    StudentClass(const string &inName,
                 const int inId);
    string getName() const;
    int getId() const;
};
//Global function prototypes are here
int main()
{
  StudentClass sl("Pikachu", 4827);
  printStudentInfo(sl);
  return 0;
}
StudentClass::StudentClass(
     const string &inName,
     const int inId
     )
{
  name = inName;
  id = inId;
}
```

```
int StudentClass::getId(
     ) const
{
  return id;
}

string StudentClass::getName(
     ) const
{
  return name;
}

void printStudentInfo(
     const StudentClass &student
     )
{
  cout << "Name: ";
  cout << student.getName();
  cout << " Id: " << student.getId() << endl;
}
```

Name: Pikachu Id: 4827

Note: The keyword const AFTER the closing paren of the parameter list for a member function means that the function will not modify any data members of the object the function was called on (i.e. before the dot operator in the calling function)

Better All Around – Make print a Member

```
class StudentClass
{
  public:
    string name;
    int    id;

  public:
    StudentClass(const string &inName,
                 const int inId);
    void printInfo() const;
};

int main()
{
  StudentClass s1("Pikachu", 4827);
  s1.printInfo();
  return 0;
}
```

```
StudentClass::StudentClass(
     const string &inName,
     const int inId
     )
{
  name = inName;
  id = inId;
}

void StudentClass::printInfo() const
{
  cout << "Name: " << name <<
          " Id: " << id << endl;
}
```

Name: Pikachu Id: 4827

Static Data Members

- Recall static variables in global functions
  - Variable stored in global space, and exists throughout program
- Static data members in classes have a similar meaning
  - A static data member only is stored in memory ***once*** per class
    - Not once per object like non-static variables!
  - All objects of the class, and in fact the class itself, "shares" this one instance of static data members
  - Static data members are initialized outside of class member functions
    - This is true even if static member is private!
    - Static data member are only allowed to be initialized once
    - Scope resolution is used to indicate a static member is being initialized
- Often used to count objects created, destroyed, or function calls

Static Data Member Example

```
class MiscClass
{
  private:
    static int numObjs;
    int id;
  public:
    MiscClass()
    {
      id = numObjs;
      numObjs++;
    }
    void printInfo() const
    { cout << "Misc Id: " << id << endl; }
};

int MiscClass::numObjs = 0;

int main()
{
  MiscClass m1, m2, m3;
  MiscClass miscObj;
  MiscClass temp;
  m1.printInfo();
  m2.printInfo();
  m3.printInfo();
  miscObj.printInfo();
  temp.printInfo();
  return 0;
}
```

Static data member. Even though 5 MiscClass objects were created below, and each has its own id associate with it, all 5 share the ONE static member "numObjs".

Since a ctor is called every time an object is created, the id for the new object is assigned using the value of the shared static variable, and it is updated.

Note: Avoid horrible style like this!!!!

This is the syntax for initializing a static data member. The data member is private, but initialization must be done outside the class, as shown.

```
Misc Id: 0
Misc Id: 1
Misc Id: 2
Misc Id: 3
Misc Id: 4
```


Public Static Data Members

```
class NewMiscClass
{
  private:
    int id;
  public:
    static int num;
  public:
    NewMiscClass()
    {
      id = num;
      num++;
    }

    void printInfo() const
    {
      cout << "NewMisc Id: " <<
              id << endl;
    }
};
```

```
int NewMiscClass::numObjs = 0;

int main()
{
  NewMiscClass m1, m2, m3;
  NewMiscClass miscObj;
  NewMiscClass temp;

  m1.printInfo();
  m2.printInfo();
  m3.printInfo();
  miscObj.printInfo();
  temp.printInfo();

  cout << "Next ID: " << m1.num << endl;
  cout << "Next ID: " << temp.num << endl;
  cout << "Next ID: " << NewMiscClass::num << endl;
  return 0;
}
```

```
NewMisc Id: 0
NewMisc Id: 1
NewMisc Id: 2
NewMisc Id: 3
NewMisc Id: 4
Next ID: 5
Next ID: 5
Next ID: 5
```

When the static member is public, it can be accessed using and individual object.

More appropriately and commonly, the static member can be accessed using the class name and scope resolution. This makes sense, since the static member doesn't belong to an object, but rather the class as a whole.

Static Member Functions

- A static member function also "belongs to a class"
- Static member functions can access private (or public) static data member variables
  - A static member function can NOT access non-static data members!
- Static functions can be called using the class name and scope resolution
- Allows static data members to be private
  - Public data members are essentially global and should be avoided

Static Member Function, Example

```
class NewMiscClass
{
  private:
    int id;
    static int numObjs;
  public:
    NewMiscClass()
    {
      id = numObjs;
      numObjs++;
    }

    void printInfo() const
    {
      cout << "NewMisc Id: " <<
              id << endl;
    }

    static void printNumObjects()
    {
      cout << "Num Objs: " <<
              numObjs << endl;
    }
};
```

```
int NewMiscClass::numObjs = 0;

int main()
{
  NewMiscClass m1, m2, m3;
  NewMiscClass miscObj;
  NewMiscClass temp;

  m1.printInfo();
  m2.printInfo();
  m3.printInfo();
  miscObj.printInfo();
  temp.printInfo();

  //cout << "Next ID: " << m1.numObjs << endl;
  //cout << "Next ID: " << temp.numObjs << endl;
  //cout << "Next ID: " << NewMiscClass::numObjs << endl;
  m1.printNumObjects();
  temp.printNumObjects();
  NewMiscClass::printNumObjects();
  return 0;
}
```

```
NewMisc Id: 0
NewMisc Id: 1
NewMisc Id: 2
NewMisc Id: 3
NewMisc Id: 4
Next ID: 5
Next ID: 5
Next ID: 5
```

Commented lines would now cause compiler errors (since numObjs is private). Can access via the static member function, however.

Inline Functions

- Recall from discussion of compiling vs. linking
  - Compiler leaves "holes" in place of function calls
  - Linker fills in holes with address of function later
- When the compiler comes across a call to an "inline function", it ***may*** replace the function call with the function body
  - The function body must be in scope so the compiler knows what the function body contains
  - Allows the programmer to use function calls, without the loss of efficiency associated with a function call
  - The compiler may choose not to inline a function, even when requested by the programmer

Inline Global Functions

- Request a global function to be inlined using the keyword "inline" before the function prototype and header

```
inline int add2(int val);

int main()
{
  int x;

  x = add2(10);

  cout << x << endl;
  return 0;
}

inline int add2(int val)
{
  return val + 2;
}
```

For the program on the left, a compiler might generate object code that would correspond to one of the programs on the right.

```
int main()
{
  int x;

  x = 10 + 2;

  cout << x << endl;
  return 0;
}
```

```
int main()
{
  int x;

  x = 12;

  cout << x << endl;
  return 0;
}
```


Inline Member Functions

- Request a member function to be inlined by providing the function body within the class definition
  - Recall this is usually poor design, as it does not separate the implementation from the interface
  - Generally only ***very short*** member functions (i.e. one or two statements) are requested for inlining
- Constructor, getLeftVal, getOper, and getRightVal ***may*** be inlined
- performOperation will not be inlined

```
class OperationClass
{
  private:
    int leftVal;
    char oper;
    int rightVal;

  public:
    OperationClass(int l, char o, int r):
            leftVal(l), oper(o), rightVal(r)
    { ; }
    int getLeftVal() const
    {
      return leftVal;
    }
    char getOper() const
    {
      return oper;
    }
    int getRightVal() const
    {
      return rightVal;
    }
    int performOperation() const;
};
```


Inline Member Functions Example Continued

```
int main()
{
  OperationClass op1(5, '+', 2);
  OperationClass op2(8, '-', 2);

  cout << op1.getLeftVal() << op1.getOper() << op1.getRightVal() <<
          "=" << op1.performOperation() << endl;
  cout << op2.getLeftVal() << op2.getOper() << op2.getRightVal() <<
          "=" << op2.performOperation() << endl;
  return 0;
}

int OperationClass::performOperation(
    ) const
{
  int result;
  if (oper == '+')
  {
    result = leftVal + rightVal;
  }
  else if (oper == '-')
  {
    result = leftVal - rightVal;
  }
  return result;
}
```

```
5+2=7
8-2=6
```


Designing Classes

- What should be a class? What should not?
  - Remember, OOP is beneficial because it can be used to generate programs that look and work like the "real world"
  - Create classes to group data and functionality for "things" that will be used in the program
  - "Actions" should be designed and implemented as functions, not classes
    - For example, CardClass, DeckClass, BankRollClass, etc
    - ***NOT*** DealCardClass, CalculateWinnerClass, etc
- What should be member variables? What should not?
  - Member variables should be data that will describe attributes of all objects of the class
  - Values that don't describe attributes of objects should ***not*** be members
    - For example, if you notice that every member function of a class uses a variable named "i" as a loop variable, you may be tempted to make it a member variable, so it doesn't have be declared in every individual function. This would be a ***poor*** design, however, since the loop variable "i" does not describe an attribute of the objects of the class!