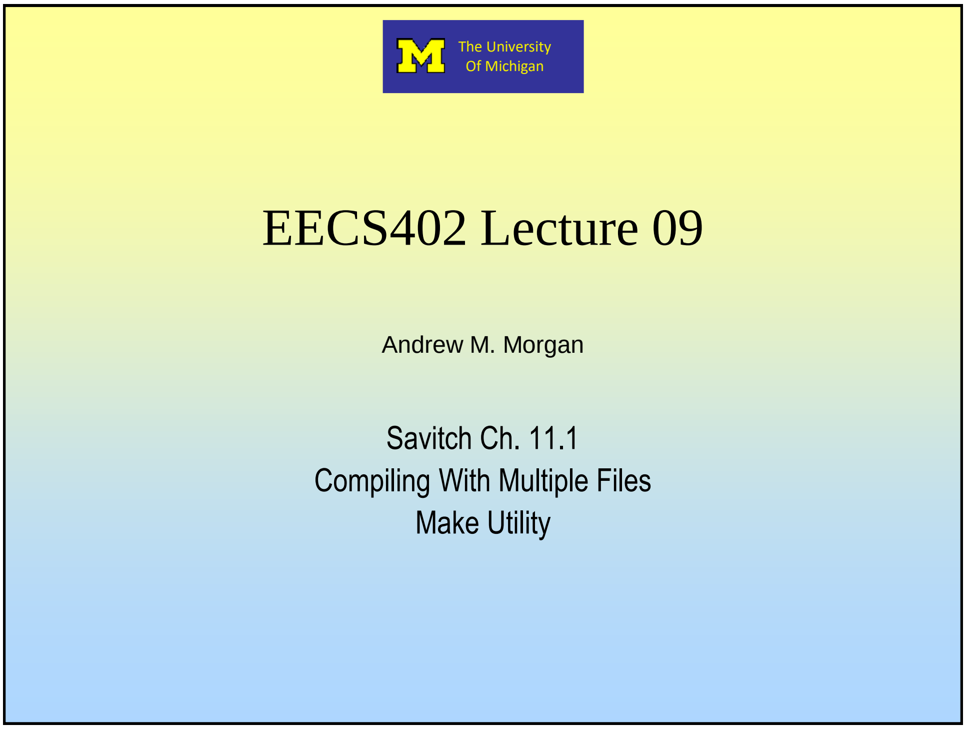
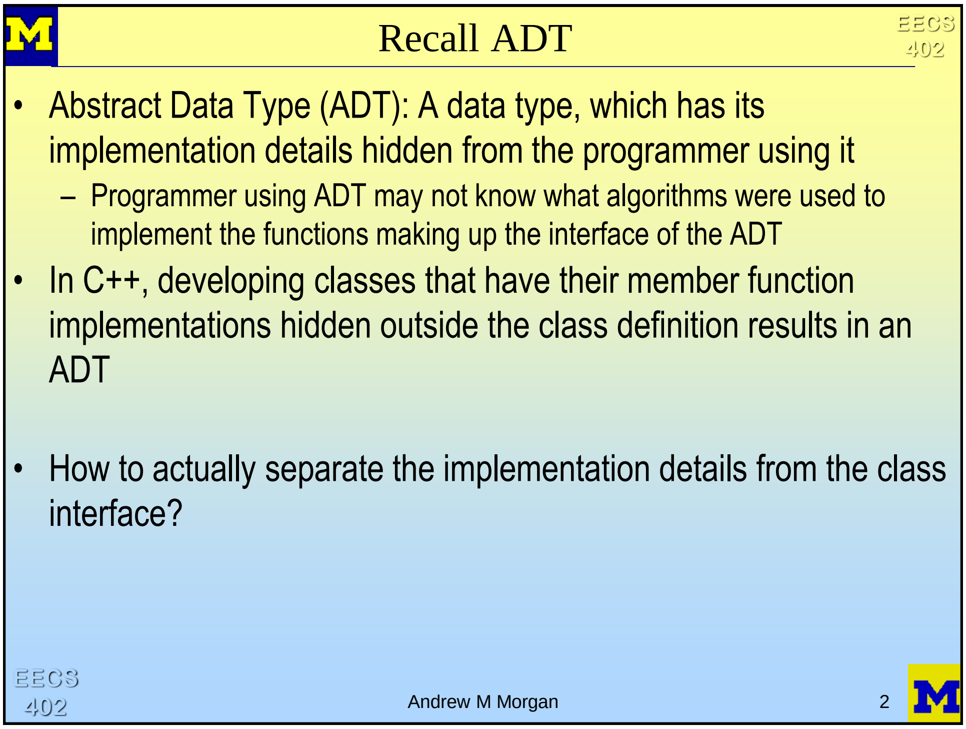


The University Of Michigan

EECS402 Lecture 09

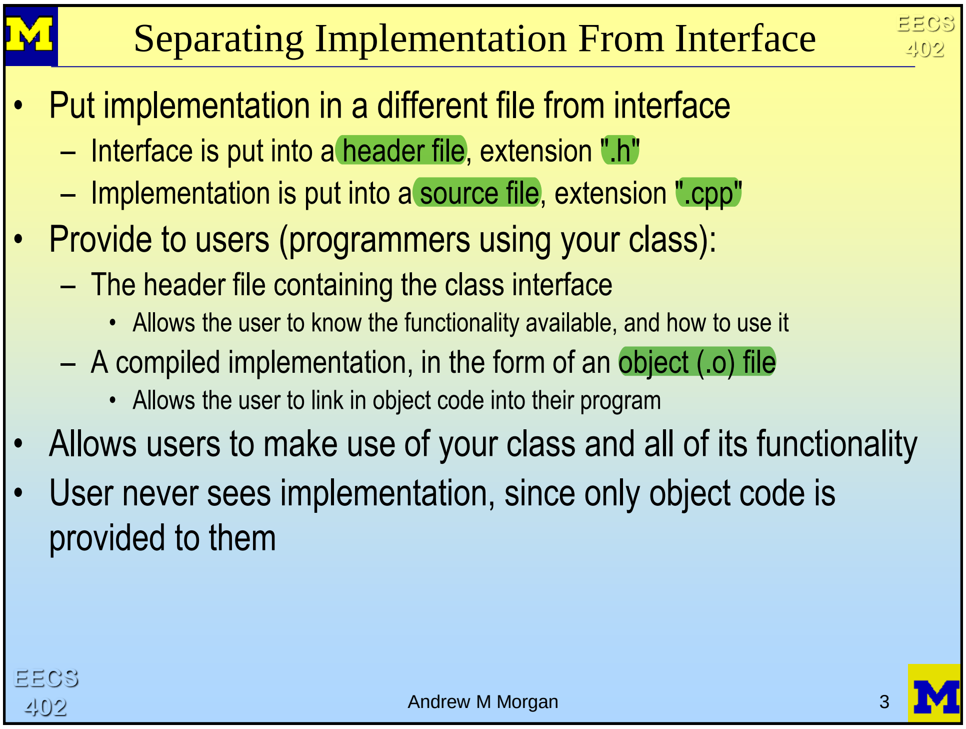
Andrew M. Morgan

Savitch Ch. 11.1
Compiling With Multiple Files
Make Utility



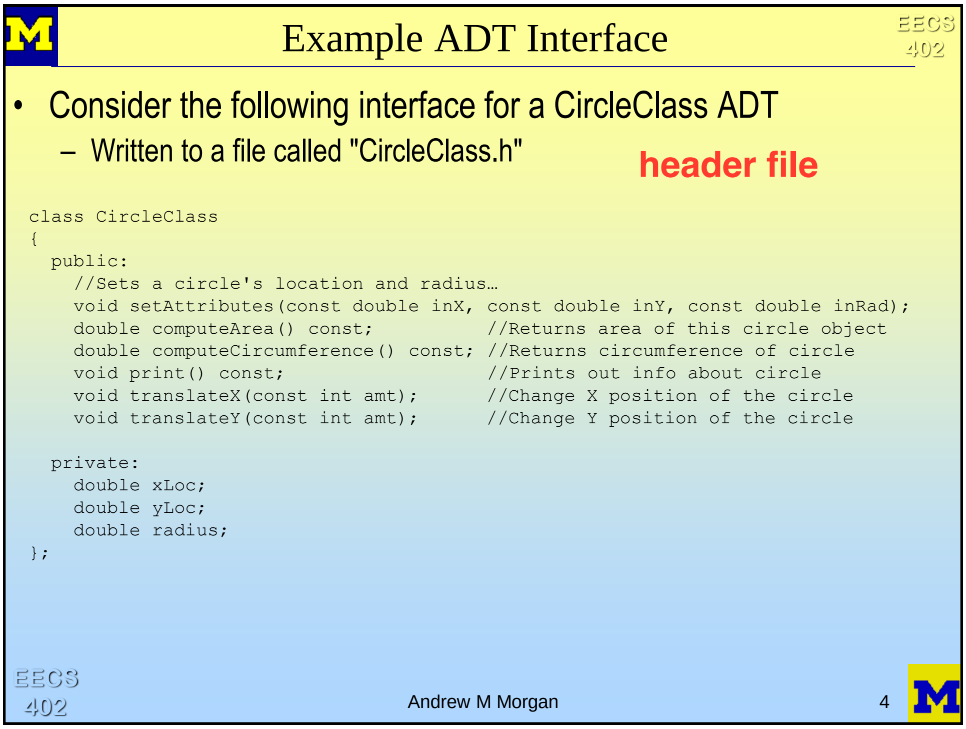
Recall ADT

- Abstract Data Type (ADT): A data type, which has its implementation details hidden from the programmer using it
  - Programmer using ADT may not know what algorithms were used to implement the functions making up the interface of the ADT
- In C++, developing classes that have their member function implementations hidden outside the class definition results in an ADT
- How to actually separate the implementation details from the class interface?



Separating Implementation From Interface

- Put implementation in a different file from interface
  - Interface is put into a header file, extension ".h"
  - Implementation is put into a source file, extension ".cpp"
- Provide to users (programmers using your class):
  - The header file containing the class interface
    - Allows the user to know the functionality available, and how to use it
  - A compiled implementation, in the form of an object (.o) file
    - Allows the user to link in object code into their program
- Allows users to make use of your class and all of its functionality
- User never sees implementation, since only object code is provided to them

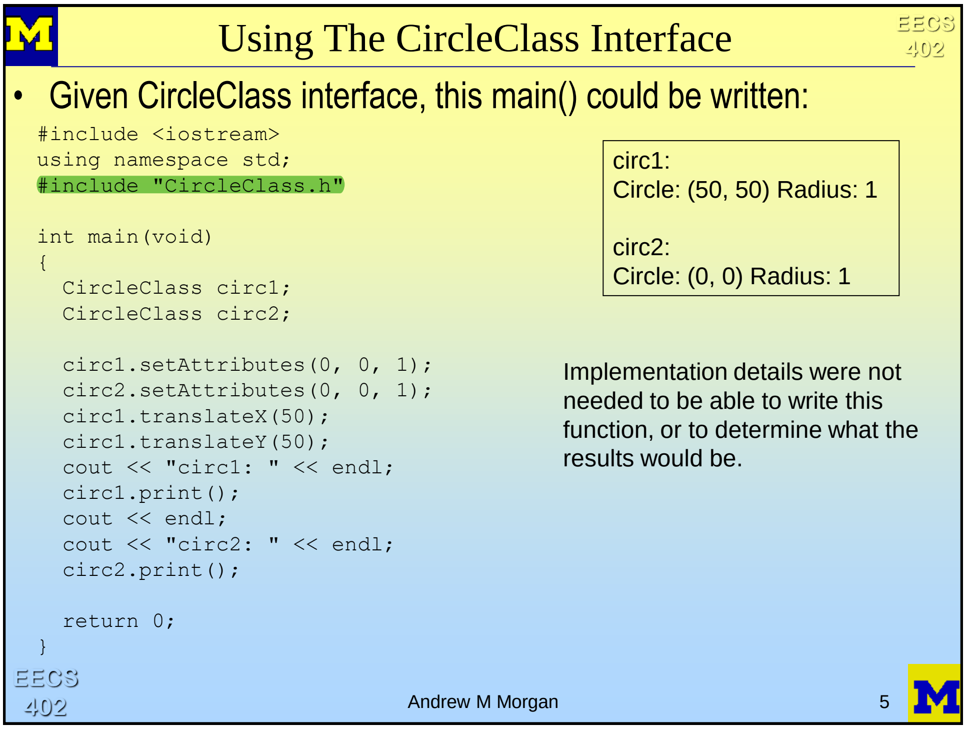


Example ADT Interface

- Consider the following interface for a CircleClass ADT
  - Written to a file called "CircleClass.h"

header file

```
class CircleClass
{
  public:
    //Sets a circle's location and radius…
    void setAttributes(const double inX, const double inY, const double inRad);
    double computeArea() const;          //Returns area of this circle object
    double computeCircumference() const; //Returns circumference of circle
    void print() const;                  //Prints out info about circle
    void translateX(const int amt);      //Change X position of the circle
    void translateY(const int amt);      //Change Y position of the circle

  private:
    double xLoc;
    double yLoc;
    double radius;
};
```



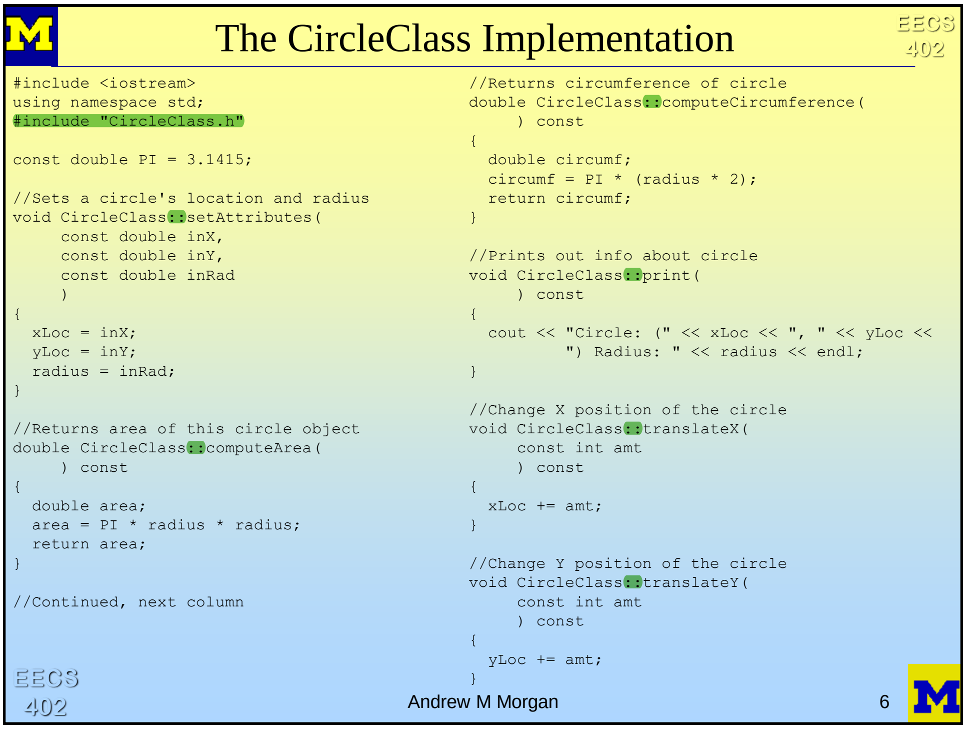
Using The CircleClass Interface

- Given CircleClass interface, this main() could be written:

```
#include <iostream>
using namespace std;
#include "CircleClass.h"

int main(void)
{
  CircleClass circ1;
  CircleClass circ2;

  circ1.setAttributes(0, 0, 1);
  circ2.setAttributes(0, 0, 1);
  circ1.translateX(50);
  circ1.translateY(50);
  cout << "circ1: " << endl;
  circ1.print();
  cout << endl;
  cout << "circ2: " << endl;
  circ2.print();

  return 0;
}
```

```
circ1:
Circle: (50, 50) Radius: 1

circ2:
Circle: (0, 0) Radius: 1
```

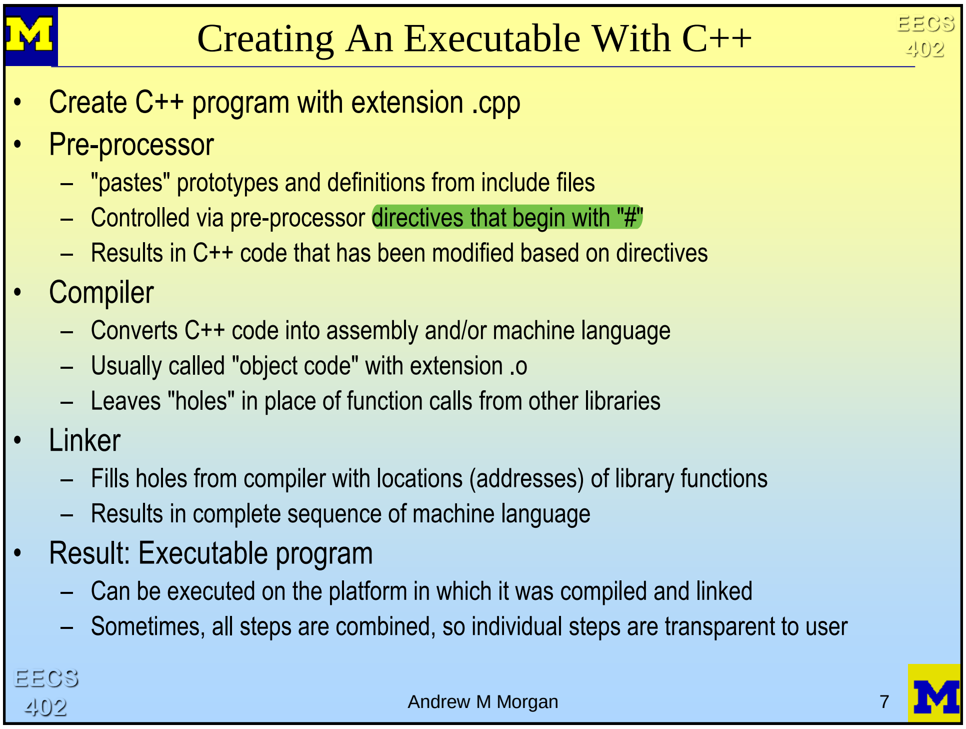
Implementation details were not needed to be able to write this function, or to determine what the results would be.



The CircleClass Implementation

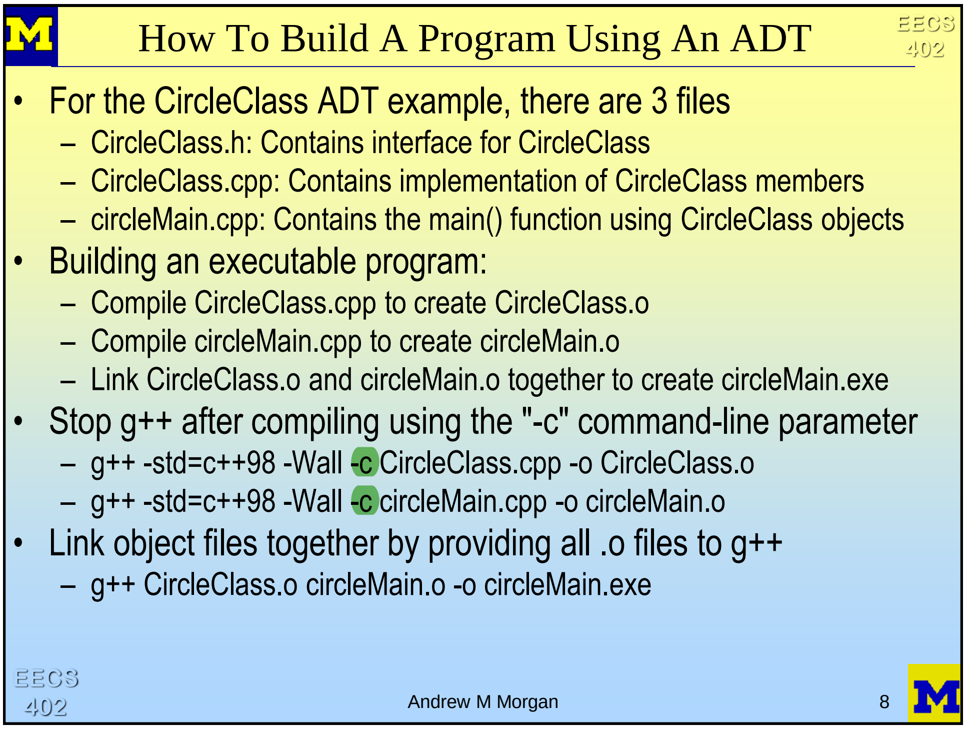
```
#include <iostream>
using namespace std;
#include "CircleClass.h"

const double PI = 3.1415;

//Sets a circle's location and radius
void CircleClass::setAttributes(
     const double inX,
     const double inY,
     const double inRad
     )
{
  xLoc = inX;
  yLoc = inY;
  radius = inRad;
}

//Returns area of this circle object
double CircleClass::computeArea(
     ) const
{
  double area;
  area = PI * radius * radius;
  return area;
}

//Continued, next column
```

```
//Returns circumference of circle
double CircleClass::computeCircumference(
     ) const
{
  double circumf;
  circumf = PI * (radius * 2);
  return circumf;
}

//Prints out info about circle
void CircleClass::print(
     ) const
{
  cout << "Circle: (" << xLoc << ", " << yLoc <<
          ") Radius: " << radius << endl;
}

//Change X position of the circle
void CircleClass::translateX(
     const int amt
     ) const
{
  xLoc += amt;
}

//Change Y position of the circle
void CircleClass::translateY(
     const int amt
     ) const
{
  yLoc += amt;
}
```



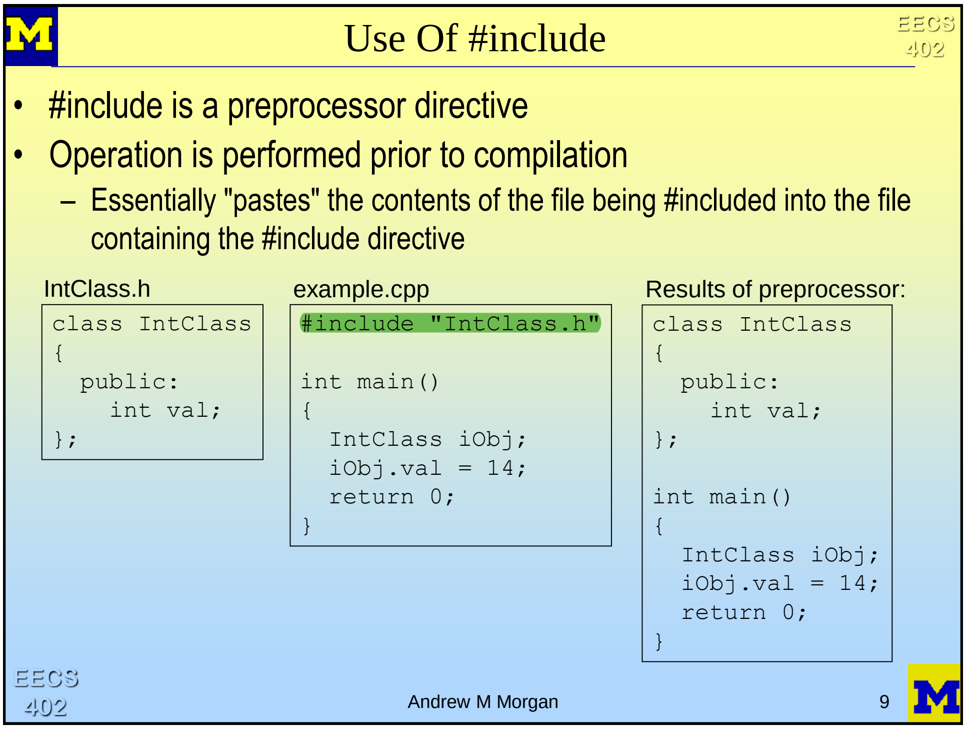
Creating An Executable With C++

- Create C++ program with extension .cpp
- Pre-processor
  - "pastes" prototypes and definitions from include files
  - Controlled via pre-processor directives that begin with "#"
  - Results in C++ code that has been modified based on directives
- Compiler
  - Converts C++ code into assembly and/or machine language
  - Usually called "object code" with extension .o
  - Leaves "holes" in place of function calls from other libraries
- Linker
  - Fills holes from compiler with locations (addresses) of library functions
  - Results in complete sequence of machine language
- Result: Executable program
  - Can be executed on the platform in which it was compiled and linked
  - Sometimes, all steps are combined, so individual steps are transparent to user



How To Build A Program Using An ADT

- For the CircleClass ADT example, there are 3 files
  - CircleClass.h: Contains interface for CircleClass
  - CircleClass.cpp: Contains implementation of CircleClass members
  - circleMain.cpp: Contains the main() function using CircleClass objects
- Building an executable program:
  - Compile CircleClass.cpp to create CircleClass.o
  - Compile circleMain.cpp to create circleMain.o
  - Link CircleClass.o and circleMain.o together to create circleMain.exe
- Stop g++ after compiling using the "-c" command-line parameter
  - g++ -std=c++98 -Wall -c CircleClass.cpp -o CircleClass.o
  - g++ -std=c++98 -Wall -c circleMain.cpp -o circleMain.o
- Link object files together by providing all .o files to g++
  - g++ CircleClass.o circleMain.o -o circleMain.exe



Use Of #include

- #include is a preprocessor directive
- Operation is performed prior to compilation
  - Essentially "pastes" the contents of the file being #included into the file containing the #include directive

IntClass.h

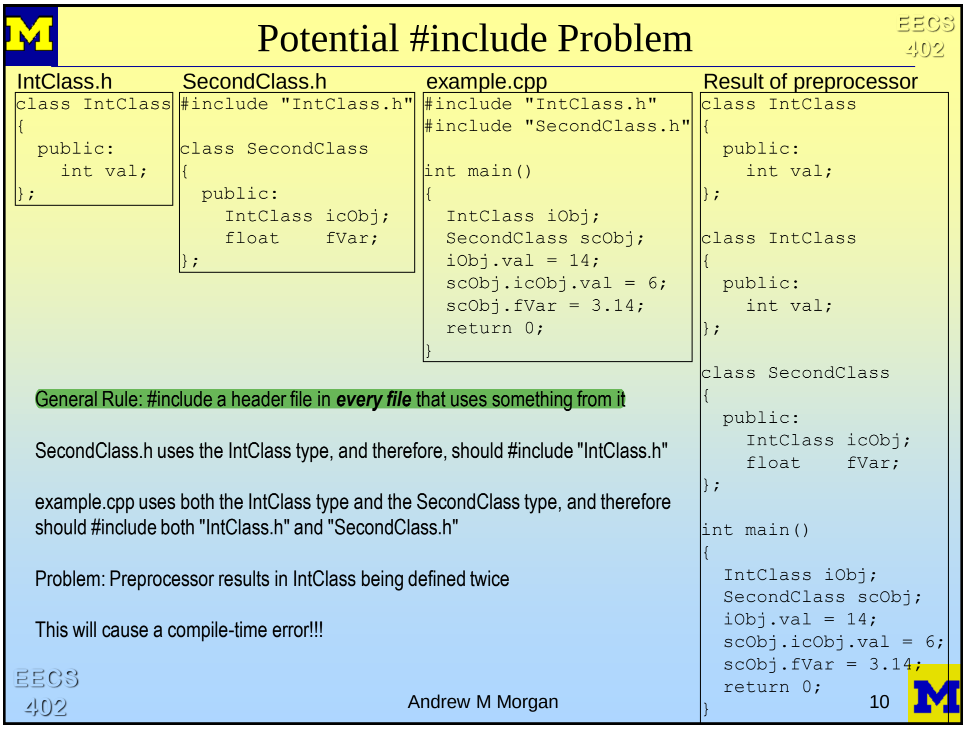
```
class IntClass
{
  public:
    int val;
};
```

example.cpp

```
#include "IntClass.h"

int main()
{
  IntClass iObj;
  iObj.val = 14;
  return 0;
}
```

Results of preprocessor:

```
class IntClass
{
  public:
    int val;
};

int main()
{
  IntClass iObj;
  iObj.val = 14;
  return 0;
}
```



Potential #include Problem

IntClass.h

```
class IntClass
{
  public:
    int val;
};
```

SecondClass.h

```
#include "IntClass.h"

class SecondClass
{
  public:
    IntClass icObj;
    float    fVar;
};
```

example.cpp

```
#include "IntClass.h"
#include "SecondClass.h"

int main()
{
  IntClass iObj;
  SecondClass scObj;
  iObj.val = 14;
  scObj.icObj.val = 6;
  scObj.fVar = 3.14;
  return 0;
}
```

Result of preprocessor

```
class IntClass
{
  public:
    int val;
};

class IntClass
{
  public:
    int val;
};

class SecondClass
{
  public:
    IntClass icObj;
    float    fVar;
};

int main()
{
  IntClass iObj;
  SecondClass scObj;
  iObj.val = 14;
  scObj.icObj.val = 6;
  scObj.fVar = 3.14;
  return 0;
}
```

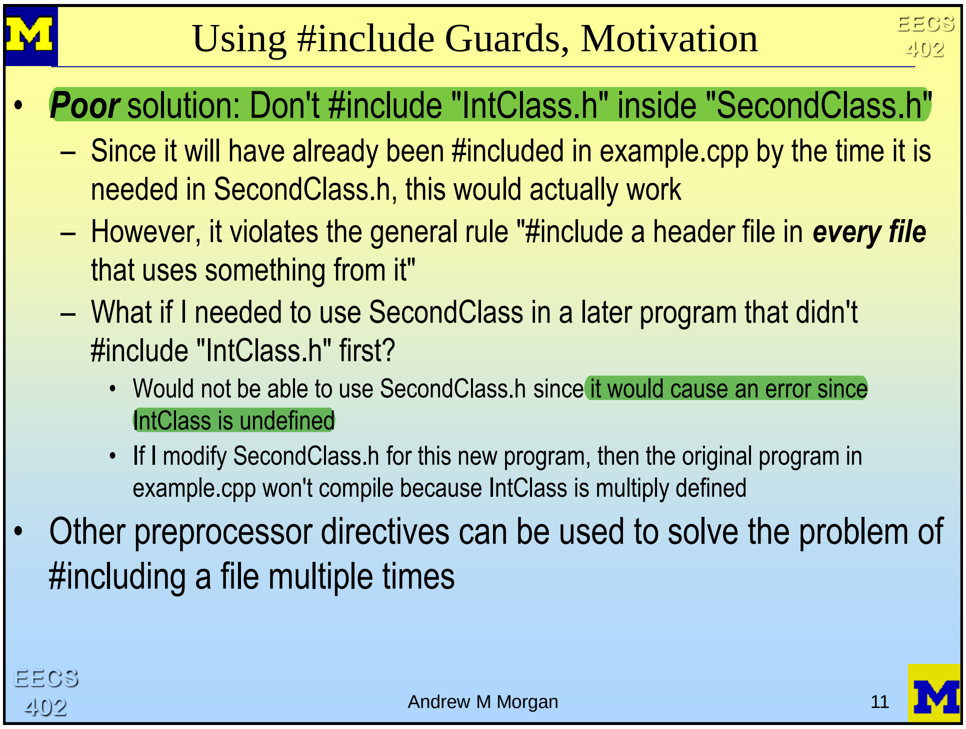
General Rule: #include a header file in ***every file*** that uses something from it

SecondClass.h uses the IntClass type, and therefore, should #include "IntClass.h"

example.cpp uses both the IntClass type and the SecondClass type, and therefore should #include both "IntClass.h" and "SecondClass.h"

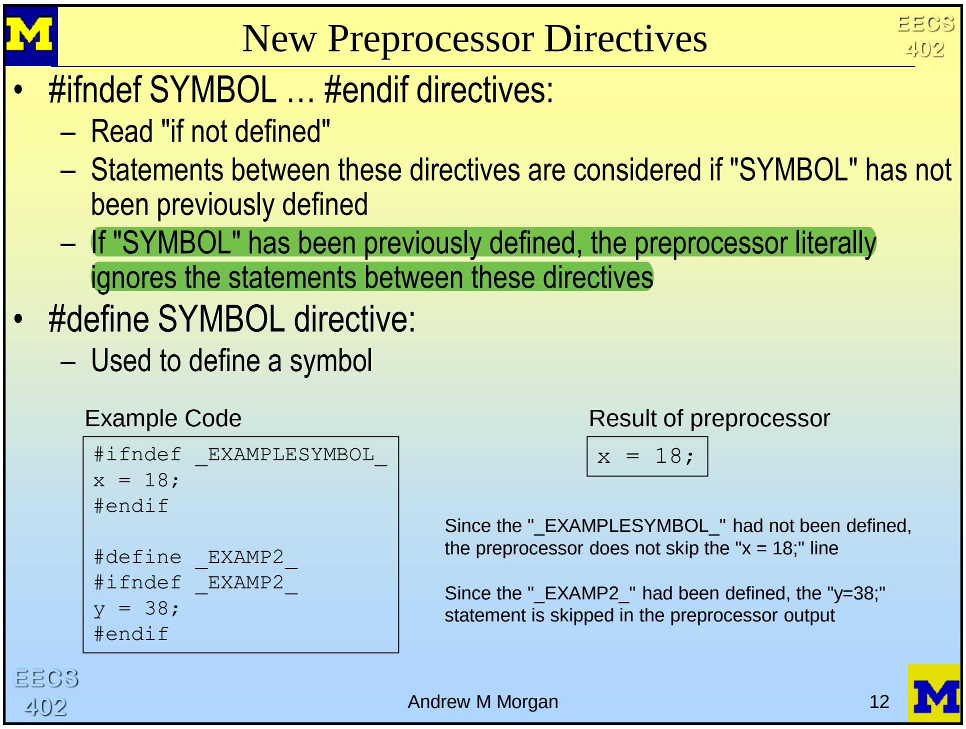
Problem: Preprocessor results in IntClass being defined twice

This will cause a compile-time error!!!



Using #include Guards, Motivation

- ***Poor*** solution: Don't #include "IntClass.h" inside "SecondClass.h"
  - Since it will have already been #included in example.cpp by the time it is needed in SecondClass.h, this would actually work
  - However, it violates the general rule "#include a header file in ***every file*** that uses something from it"
  - What if I needed to use SecondClass in a later program that didn't #include "IntClass.h" first?
    - Would not be able to use SecondClass.h since it would cause an error since IntClass is undefined
    - If I modify SecondClass.h for this new program, then the original program in example.cpp won't compile because IntClass is multiply defined
- Other preprocessor directives can be used to solve the problem of #including a file multiple times



New Preprocessor Directives

- #ifndef SYMBOL … #endif directives:
  - Read "if not defined"
  - Statements between these directives are considered if "SYMBOL" has not been previously defined
  - If "SYMBOL" has been previously defined, the preprocessor literally ignores the statements between these directives
- #define SYMBOL directive:
  - Used to define a symbol

Example Code

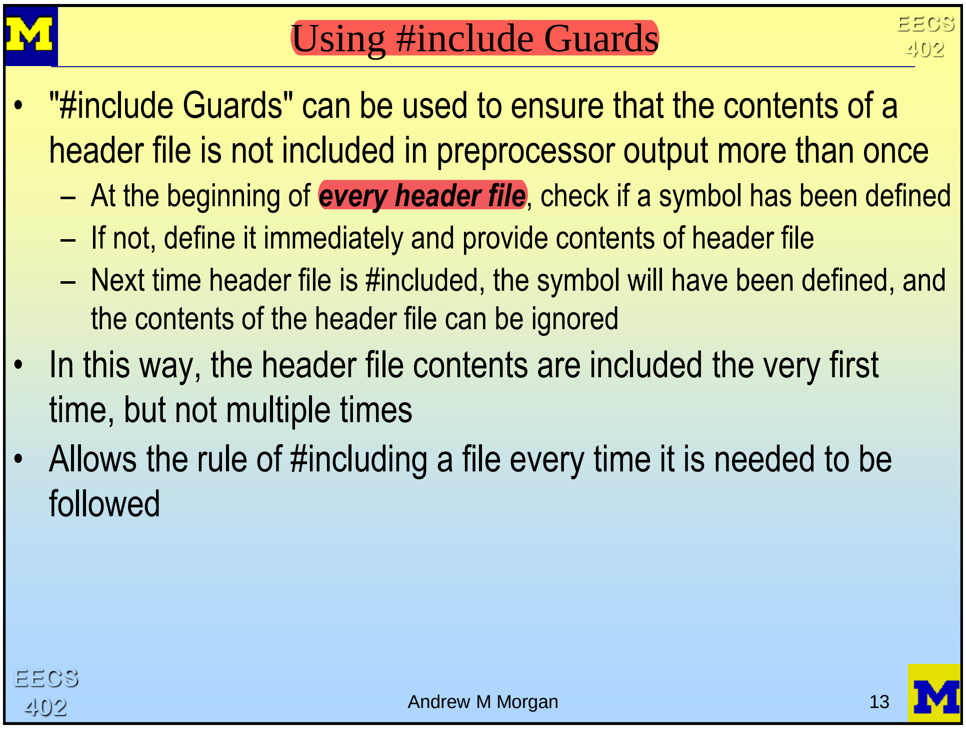
```
#ifndef _EXAMPLESYMBOL_
x = 18;
#endif

#define _EXAMP2_
#ifndef _EXAMP2_
y = 38;
#endif
```

Result of preprocessor

```
x = 18;
```

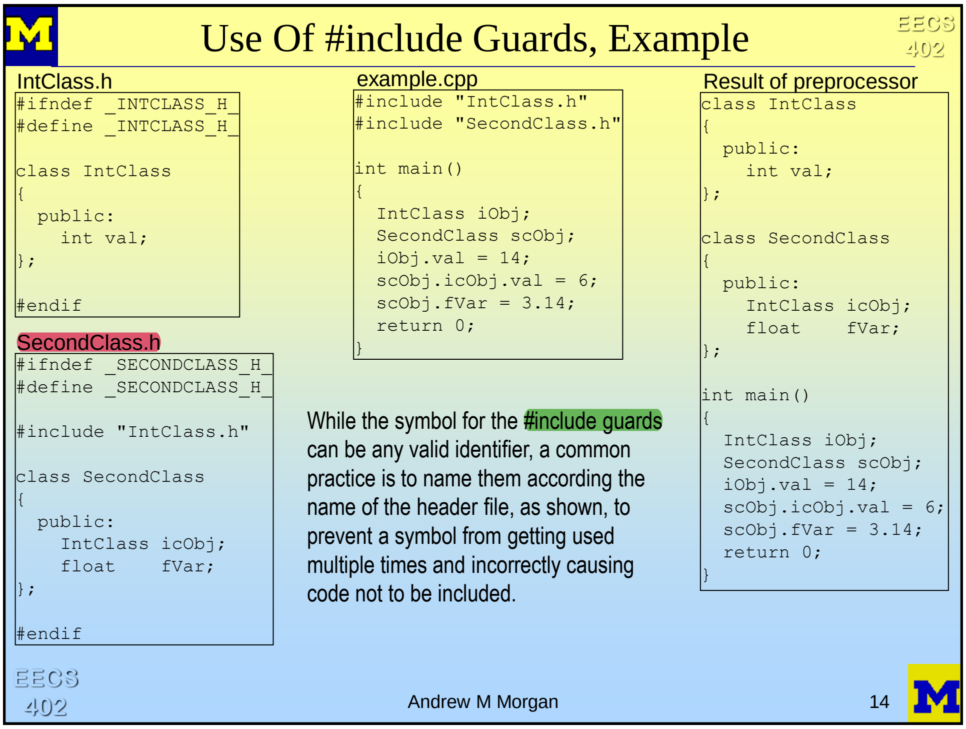
Since the "_EXAMPLESYMBOL_" had not been defined, the preprocessor does not skip the "x = 18;" line

Since the "_EXAMP2_" had been defined, the "y=38;" statement is skipped in the preprocessor output



Using #include Guards

- "#include Guards" can be used to ensure that the contents of a header file is not included in preprocessor output more than once
  - At the beginning of ***every header file***, check if a symbol has been defined
  - If not, define it immediately and provide contents of header file
  - Next time header file is #included, the symbol will have been defined, and the contents of the header file can be ignored
- In this way, the header file contents are included the very first time, but not multiple times
- Allows the rule of #including a file every time it is needed to be followed



Use Of #include Guards, Example

IntClass.h

```
#ifndef _INTCLASS_H_
#define _INTCLASS_H_

class IntClass
{
  public:
    int val;
};

#endif
```

SecondClass.h

```
#ifndef _SECONDCLASS_H_
#define _SECONDCLASS_H_

#include "IntClass.h"

class SecondClass
{
  public:
    IntClass icObj;
    float    fVar;
};

#endif
```

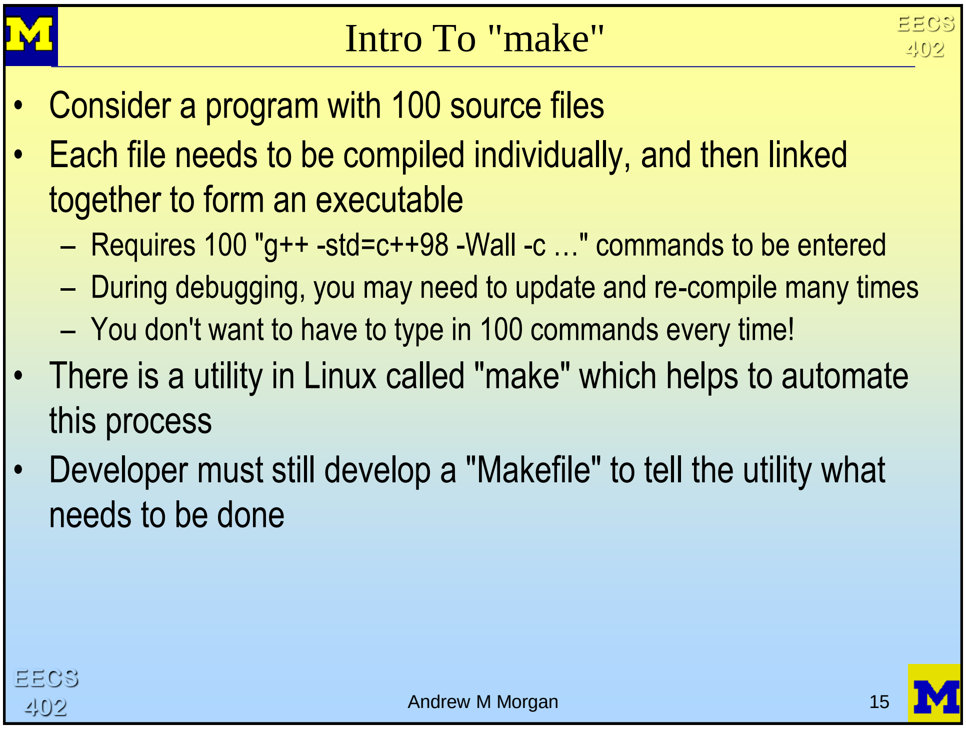
example.cpp

```
#include "IntClass.h"
#include "SecondClass.h"

int main()
{
  IntClass iObj;
  SecondClass scObj;
  iObj.val = 14;
  scObj.icObj.val = 6;
  scObj.fVar = 3.14;
  return 0;
}
```

While the symbol for the #include guards can be any valid identifier, a common practice is to name them according the name of the header file, as shown, to prevent a symbol from getting used multiple times and incorrectly causing code not to be included.

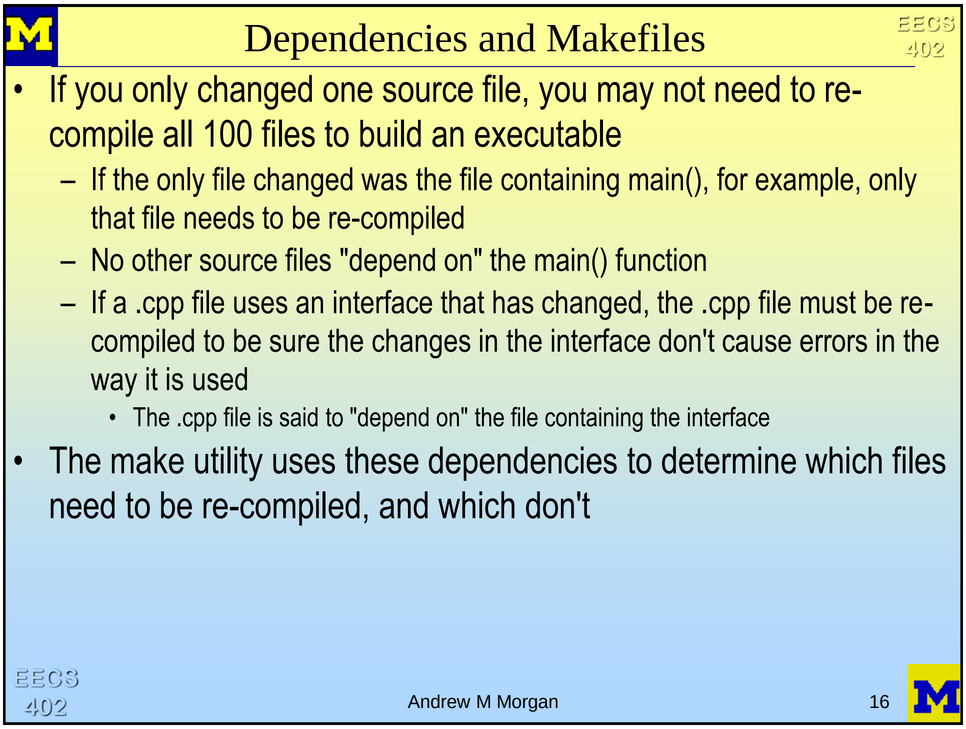
Result of preprocessor

```
class IntClass
{
  public:
    int val;
};

class SecondClass
{
  public:
    IntClass icObj;
    float    fVar;
};

int main()
{
  IntClass iObj;
  SecondClass scObj;
  iObj.val = 14;
  scObj.icObj.val = 6;
  scObj.fVar = 3.14;
  return 0;
}
```



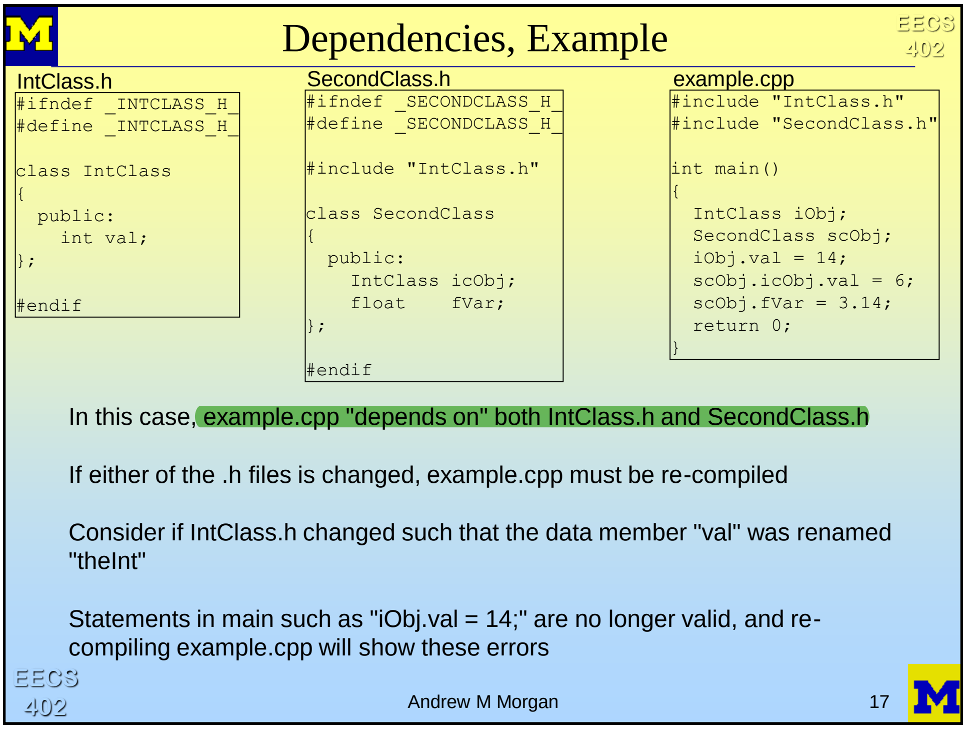
Intro To "make"

- Consider a program with 100 source files
- Each file needs to be compiled individually, and then linked together to form an executable
  - Requires 100 "g++ -std=c++98 -Wall -c …" commands to be entered
  - During debugging, you may need to update and re-compile many times
  - You don't want to have to type in 100 commands every time!
- There is a utility in Linux called "make" which helps to automate this process
- Developer must still develop a "Makefile" to tell the utility what needs to be done



Dependencies and Makefiles

- If you only changed one source file, you may not need to re-compile all 100 files to build an executable
  - If the only file changed was the file containing main(), for example, only that file needs to be re-compiled
  - No other source files "depend on" the main() function
  - If a .cpp file uses an interface that has changed, the .cpp file must be re-compiled to be sure the changes in the interface don't cause errors in the way it is used
    - The .cpp file is said to "depend on" the file containing the interface
- The make utility uses these dependencies to determine which files need to be re-compiled, and which don't



Dependencies, Example

IntClass.h

```
#ifndef _INTCLASS_H_
#define _INTCLASS_H_

class IntClass
{
  public:
    int val;
};

#endif
```

SecondClass.h

```
#ifndef _SECONDCLASS_H_
#define _SECONDCLASS_H_

#include "IntClass.h"

class SecondClass
{
  public:
    IntClass icObj;
    float    fVar;
};

#endif
```

example.cpp

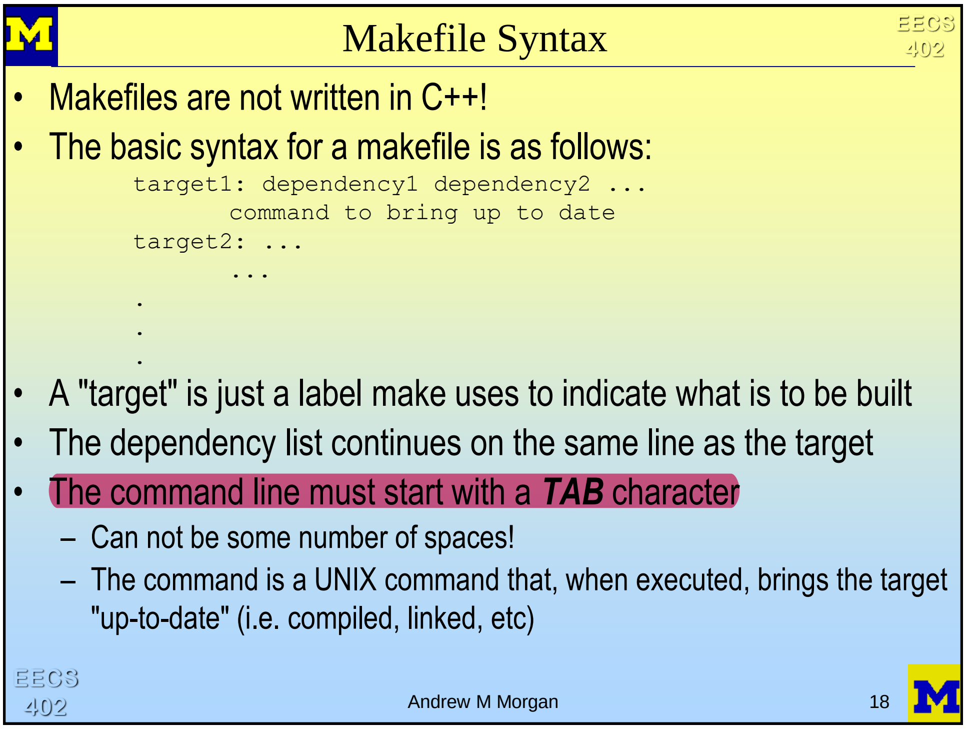
```
#include "IntClass.h"
#include "SecondClass.h"

int main()
{
  IntClass iObj;
  SecondClass scObj;
  iObj.val = 14;
  scObj.icObj.val = 6;
  scObj.fVar = 3.14;
  return 0;
}
```

In this case, example.cpp "depends on" both IntClass.h and SecondClass.h

If either of the .h files is changed, example.cpp must be re-compiled

Consider if IntClass.h changed such that the data member "val" was renamed "theInt"

Statements in main such as "iObj.val = 14;" are no longer valid, and re-compiling example.cpp will show these errors

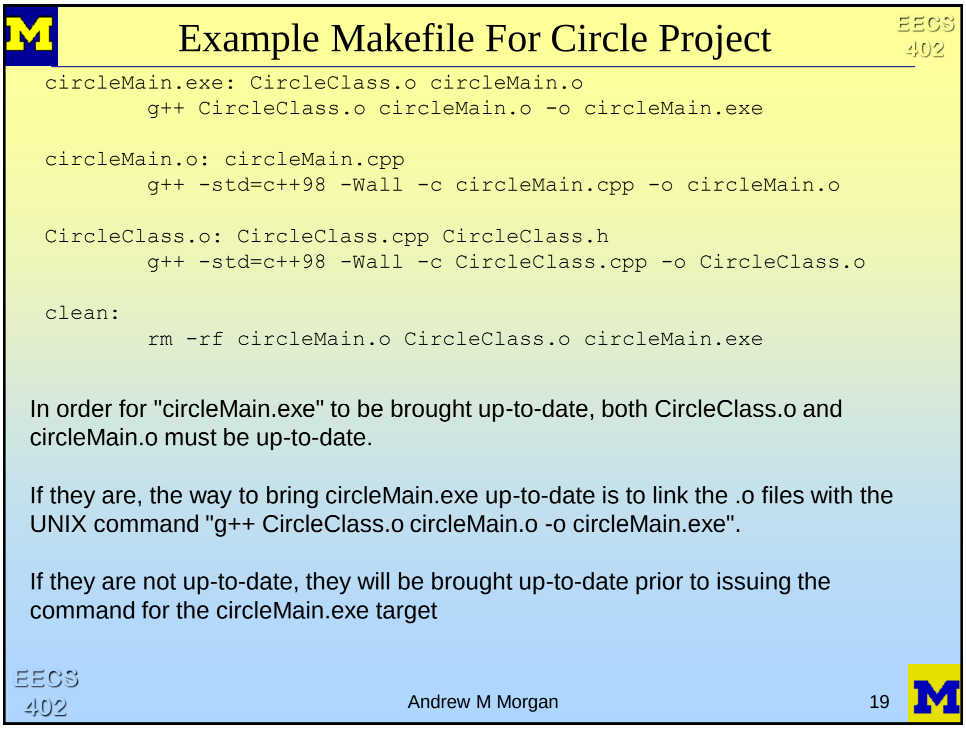


Makefile Syntax

- Makefiles are not written in C++!
- The basic syntax for a makefile is as follows:

```
target1: dependency1 dependency2 ...
         command to bring up to date
target2: ...
         ...
.
.
.
```

- A "target" is just a label make uses to indicate what is to be built
- The dependency list continues on the same line as the target
- The command line must start with a ***TAB*** character
  - Can not be some number of spaces!
  - The command is a UNIX command that, when executed, brings the target "up-to-date" (i.e. compiled, linked, etc)



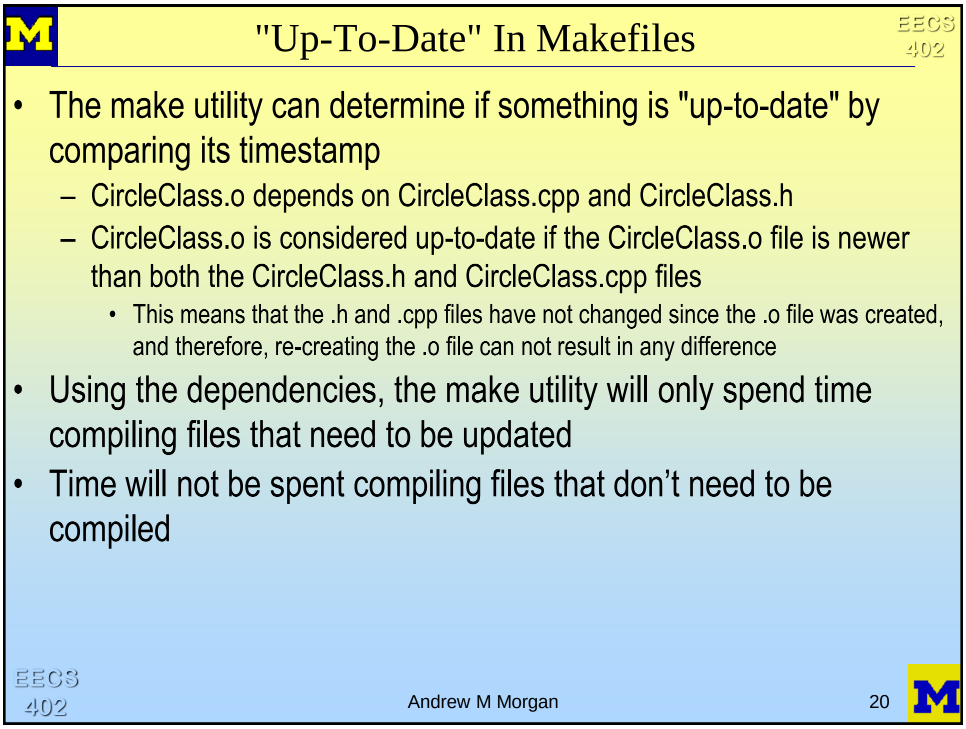
Example Makefile For Circle Project

```
circleMain.exe: CircleClass.o circleMain.o
        g++ CircleClass.o circleMain.o -o circleMain.exe

circleMain.o: circleMain.cpp
        g++ -std=c++98 -Wall -c circleMain.cpp -o circleMain.o

CircleClass.o: CircleClass.cpp CircleClass.h
        g++ -std=c++98 -Wall -c CircleClass.cpp -o CircleClass.o

clean:
        rm -rf circleMain.o CircleClass.o circleMain.exe
```

In order for "circleMain.exe" to be brought up-to-date, both CircleClass.o and circleMain.o must be up-to-date.

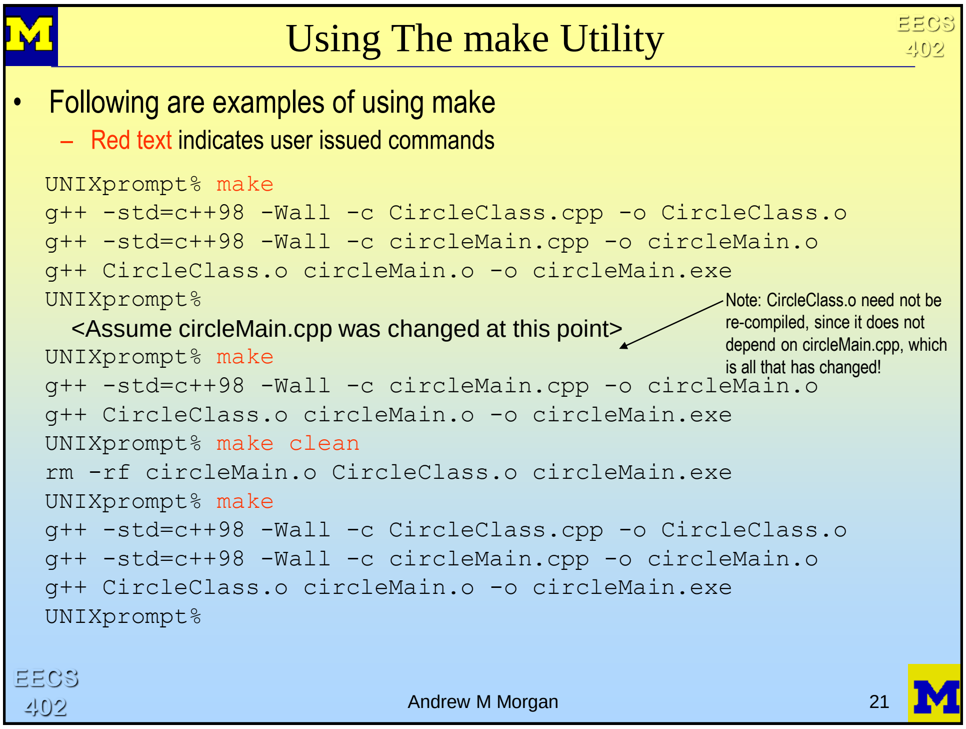
If they are, the way to bring circleMain.exe up-to-date is to link the .o files with the UNIX command "g++ CircleClass.o circleMain.o -o circleMain.exe".

If they are not up-to-date, they will be brought up-to-date prior to issuing the command for the circleMain.exe target



"Up-To-Date" In Makefiles

- The make utility can determine if something is "up-to-date" by comparing its timestamp
  - CircleClass.o depends on CircleClass.cpp and CircleClass.h
  - CircleClass.o is considered up-to-date if the CircleClass.o file is newer than both the CircleClass.h and CircleClass.cpp files
    - This means that the .h and .cpp files have not changed since the .o file was created, and therefore, re-creating the .o file can not result in any difference
- Using the dependencies, the make utility will only spend time compiling files that need to be updated
- Time will not be spent compiling files that don’t need to be compiled



EECS 402

Using The make Utility

- Following are examples of using make
  - Red text indicates user issued commands

```
UNIXprompt% make
g++ -std=c++98 -Wall -c CircleClass.cpp -o CircleClass.o
g++ -std=c++98 -Wall -c circleMain.cpp -o circleMain.o
g++ CircleClass.o circleMain.o -o circleMain.exe
UNIXprompt%
  <Assume circleMain.cpp was changed at this point>
UNIXprompt% make
g++ -std=c++98 -Wall -c circleMain.cpp -o circleMain.o
g++ CircleClass.o circleMain.o -o circleMain.exe
UNIXprompt% make clean
rm -rf circleMain.o CircleClass.o circleMain.exe
UNIXprompt% make
g++ -std=c++98 -Wall -c CircleClass.cpp -o CircleClass.o
g++ -std=c++98 -Wall -c circleMain.cpp -o circleMain.o
g++ CircleClass.o circleMain.o -o circleMain.exe
UNIXprompt%
```

Note: CircleClass.o need not be re-compiled, since it does not depend on circleMain.cpp, which is all that has changed!

EECS 402

Andrew M Morgan

21