The University Of Michigan

EECS402 Lecture 17

Andrew M. Morgan

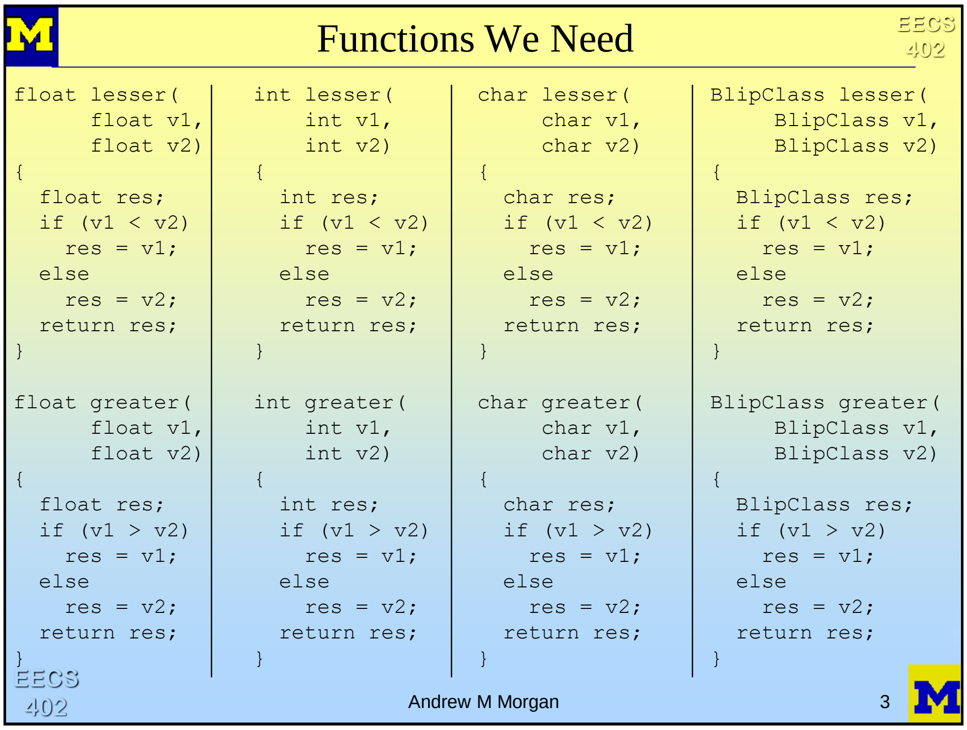
Savitch Ch. 16
Templates

Avoid duplicated codes

Consider This Program

```
int main(void)
{
  int i1 = 5, i2 = 8, i3, i4;
  float f1 = 19.2, f2 = -2.3, f3, f4;
  char c1 = 'h', c2 = 'p', c3, c4;
  BlipClass b1(6, 9), b2(7, 7), b3, b4;

  i3 = lesser(i1, i2);
  i4 = greater(i1, i2);
  f3 = lesser(f1, f2);
  f4 = greater(f1, f2);
  c3 = lesser(c1, c2);
  c4 = greater(c1, c2);
  b3 = lesser(b1, b2);
  b4 = greater(b1, b2);

  cout << "Ints:   " << i3 << " " << i4 << endl;
  cout << "Floats: " << f3 << " " << f4 << endl;
  cout << "Chars:  " << c3 << " " << c4 << endl;
  cout << "Blips:  " << b3 << " " << b4 << endl;
  return 0;
}
```

```
Ints:   5 8
Floats: -2.3 19.2
Chars:  h p
Blips:  blip: 7 7 blip: 6 9
```



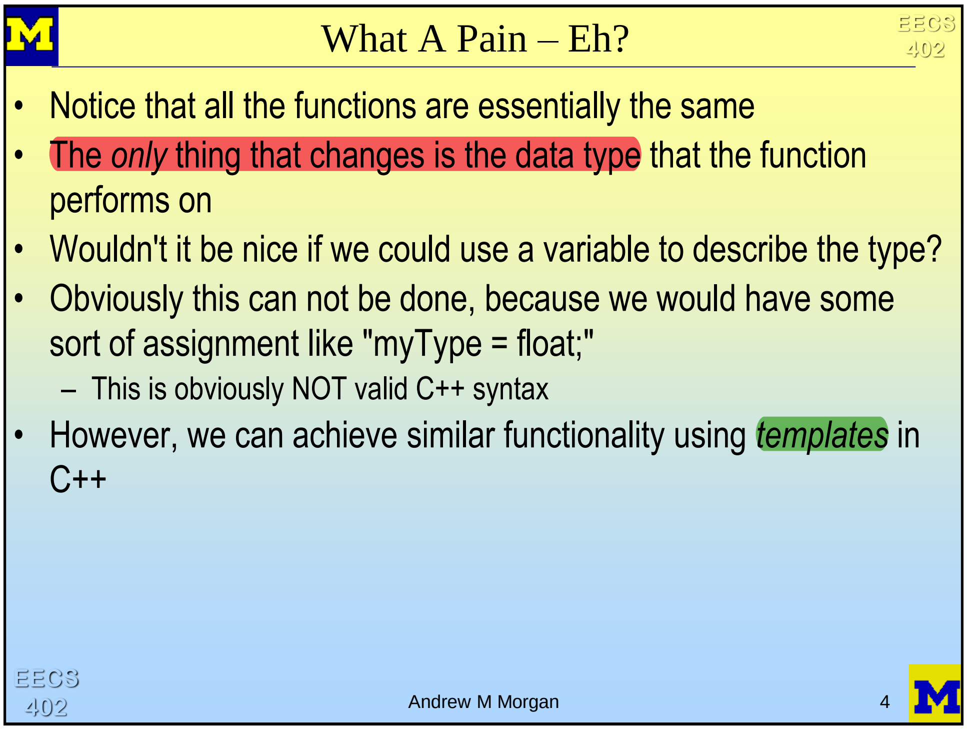
Functions We Need

```
float lesser(
      float v1,
      float v2)
{
  float res;
  if (v1 < v2)
    res = v1;
  else
    res = v2;
  return res;
}

float greater(
      float v1,
      float v2)
{
  float res;
  if (v1 > v2)
    res = v1;
  else
    res = v2;
  return res;
}
```

```
int lesser(
    int v1,
    int v2)
{
  int res;
  if (v1 < v2)
    res = v1;
  else
    res = v2;
  return res;
}

int greater(
    int v1,
    int v2)
{
  int res;
  if (v1 > v2)
    res = v1;
  else
    res = v2;
  return res;
}
```

```
char lesser(
     char v1,
     char v2)
{
  char res;
  if (v1 < v2)
    res = v1;
  else
    res = v2;
  return res;
}

char greater(
     char v1,
     char v2)
{
  char res;
  if (v1 > v2)
    res = v1;
  else
    res = v2;
  return res;
}
```

```
BlipClass lesser(
     BlipClass v1,
     BlipClass v2)
{
  BlipClass res;
  if (v1 < v2)
    res = v1;
  else
    res = v2;
  return res;
}

BlipClass greater(
     BlipClass v1,
     BlipClass v2)
{
  BlipClass res;
  if (v1 > v2)
    res = v1;
  else
    res = v2;
  return res;
}
```



What A Pain – Eh?

- Notice that all the functions are essentially the same
- The *only* thing that changes is the data type that the function performs on
- Wouldn't it be nice if we could use a variable to describe the type?
- Obviously this can not be done, because we would have some sort of assignment like "myType = float;"
  - This is obviously NOT valid C++ syntax
- However, we can achieve similar functionality using *templates* in C++

Introduction To Templates

- A template is a special construct in C++ that allows multiple types to be operated on by a single implementation of a function
- Think of a template as describing an *algorithm*
  - Since the operation we are performing is performed the same way for all different data types, it is sufficient to give code to implement an algorithm without being specific to a data type
- The word template is a keyword in C++
- A function can be "templated" - as would be the case for the lesser() and greater() functions
- A class can also be "templated"
- A templated function or class can have different data types that are left unspecified
  - In other words, multiple types can be templated

Templates, Example

The capital T is a placeholder

```
template < class T >
T lesser(T val1,
         T val2)
{
  T res;
  if (val1 < val2)
    res = val1;
  else
    res = val2;
  return res;
}

template < class T >
T greater(T val1,
          T val2)
{
  T res;
  if (val1 > val2)
    res = val1;
  else
    res = val2;
  return res;
}
```

The following two functions can replace ALL eight of the functions on slide #3.

In fact, it can even replace *automatically* any lesser() and greater() functions needed for future classes you define.

The keyword template begins a function or class that is templated.

This is followed by angle brackets, and the keyword class, and then an identifier to represent the type.

In the templated code, the type identifier replaces every instance of the data type that would change for different functions.

This template does not apply to all possible classes, we need specify this greater than in the class

Using Templates

- How do you use a templated function though?
- Just as you would any other function
- The hard work is done by the compiler
  - When passing through your program, the compiler detects which types are needed by a templated function.
  - The compiler then automatically generates a separate function for each type needed
  - The function is then compiled and is available for use, without any required actions by the user
- A programmer using a templated function may not even realize it was templated

Using Templates, Example

```
int main(void)
{
  int i1 = 5, i2 = 8, i3, i4;
  float f1 = 19.2, f2 = -2.3, f3, f4;
  char c1 = 'h', c2 = 'p', c3, c4;
  BlipClass b1(6, 9), b2(7, 7), b3, b4;

  i3 = lesser(i1, i2);
  i4 = greater(i1, i2);
  f3 = lesser(f1, f2);
  f4 = greater(f1, f2);
  c3 = lesser(c1, c2);
  c4 = greater(c1, c2);
  //b3 = lesser(b1, b2);
  //b4 = greater(b1, b2);

  cout << "Ints:   " << i3 << " " << i4 << endl;
  cout << "Floats: " << f3 << " " << f4 << endl;
  cout << "Chars:  " << c3 << " " << c4 << endl;
  //cout << "Blips:  " << b3 << " " << b4 << endl;
  return 0;
}
```

This is the exact same main() function shown earlier, when all 8 individual lesser() and greater() functions were implemented.

Currently, the BlipClass operations are commented. This is explained in the next 4 slides.

```
Ints:   5 8
Floats: -2.3 19.2
Chars:  h p
```

Assumption Made

- Very important note: What assumptions are made when you make a templated function?
- While you originally write lesser() and greater() for integers, or floats, etc, it may be used for other types
- For example, the two functions were used on the type "BlipClass" which was a user-defined class
- You MUST be sure that your user-defined classes provide any operations and/or operators that are used in the templated function
- That means the class blip must have defined an implementation for determining what ">" and "<" means for these blip objects

More On Template Assumptions

- Consider this implementation of the BlipClass:

```
class BlipClass
{
  public:
    int v1;
    int v2;

    BlipClass():v1(0),v2(0) { ; }
    BlipClass(int in1, int in2):v1(in1),v2(in2) { ; }
};
```

- The following results when trying to compile the previous program:

```
BlipClass.cpp: In function `int main()':
BlipClass.cpp:57: no match for `ostream & << BlipClass &'
BlipClass.cpp: In function `class BlipClass lesser<BlipClass>(BlipClass, BlipClass)':
BlipClass.cpp:51:   instantiated from here
BlipClass.cpp:9: no match for `BlipClass & < BlipClass &'
BlipClass.cpp: In function `class BlipClass greater<BlipClass>(BlipClass, BlipClass)':
BlipClass.cpp:52:   instantiated from here
BlipClass.cpp:21: no match for `BlipClass & > BlipClass &'
```

} Insertion operator used in main()

} Less-than operator used in templated lesser()

} Greater-than operator used in templated greater()

Updated BlipClass Implementation

Added functionality allows earlier main() to compile and be executed.

```
class BlipClass
{
  public:
    int v1;
    int v2;

    BlipClass():v1(0),v2(0) { ; }
    BlipClass(int in1, int in2):v1(in1),v2(in2) { ; }
    bool operator<(const BlipClass &rhs)
    {
      return (v1 + v2) < (rhs.v1 + rhs.v2);
    }
    bool operator>(const BlipClass &rhs)
    {
      return (v1 + v2) > (rhs.v1 + rhs.v2);
    }
};
ostream &operator<<(ostream &os, const BlipClass &rhs)
{
  os << "blip: " << rhs.v1 << " " << rhs.v2;
  return os;
}
```

Final Execution of main() Function

```
int main(void)
{
  int i1 = 5, i2 = 8, i3, i4;
  float f1 = 19.2, f2 = -2.3, f3, f4;
  char c1 = 'h', c2 = 'p', c3, c4;
  BlipClass b1(6, 9), b2(7, 7), b3, b4;

  i3 = lesser(i1, i2);
  i4 = greater(i1, i2);
  f3 = lesser(f1, f2);
  f4 = greater(f1, f2);
  c3 = lesser(c1, c2);
  c4 = greater(c1, c2);
  b3 = lesser(b1, b2);
  b4 = greater(b1, b2);

  cout << "Ints:   " << i3 << " " << i4 << endl;
  cout << "Floats: " << f3 << " " << f4 << endl;
  cout << "Chars:  " << c3 << " " << c4 << endl;
  cout << "Blips:  " << b3 << " " << b4 << endl;
  return 0;
}
```

```
Ints:   5 8
Floats: -2.3 19.2
Chars:  h p
Blips:  blip: 7 7 blip: 6 9
```

Note: Absolutely no change from the example using non-templated functions...

Templated Classes

- Templated classes are a bit different than templated functions
- The main reason for this is because a class is a *type* and objects will be declared of that type
  - Therefore, you must know what type(s) of data will be stored in the object you are creating
- This means, while a programmer may not know a *function* is templated, they must know a *class* is templated
- When creating an object of a templated class, the programmer must provide the data type during the declaration

Templated Classes, Example

```
template < class T >
class TempClass
{
  public:
    T val;
    void divide(int rhs);
};

template < class T >
void TempClass< T >::divide(int rhs)
{
  cout << val << " / " << rhs << " = ";
  cout << val / rhs << endl;
}

int main()
{
  TempClass< int > intTemp;
  TempClass< float > floatTemp;
  intTemp.val = 7;
  floatTemp.val = 7;
  intTemp.divide(3);
  floatTemp.divide(3);
  return 0;
}
```

```
7 / 3 = 2
7 / 3 = 2.33333
```

Specifies the following item (the TempClass class) will be templated

val attribute is templated to allow any type

Specifies the following item (the divide function) will be templated

Once a class is templated, it NO LONGER exists on its own – only those modified with a data type actually exist. < T > is required to specify "this is the divide function belonging to the TempClass for ints" or "floats" etc.

When an object of a templated class is declared, the type must be specified (since TempClass doesn't exist on its own)

Performs integer division

Performs floating point division

Templates With Multiple Types

- Templates can be used to template multiple data types for a class
  - Consider a data structure that maps one type to another
  - int to float: 1=>1.0, 2=>1.414, 3=>1.732. 4=>2.0, etc…
  - char to int: 'a'=>1, 'b'=>2, 'c'=>3, 'd'=>4, etc…
  - Can template both data types so one single (templated) data structure can map any type to any other type

Templating Multiple Data Types, Example

two placeholders

both are placeholders types

```
template <class DT1, class DT2>
class DataWithKey
{
  private:
    DT1 theKey; //This is the key associated with the item.
    DT2 val;    //This is the data value of the item.
  public:
    DataWithKey() { ; }
    DataWithKey(DT1 inKey, DT2 inVal):theKey(intKey), val(inVal) { ; }
    //Reader / writer functions here..
    DT1 getKey()
    { return theKey; }
    DT2 getVal()
    { return val; }
    void setInfo(DT1 inKey, DT2 inVal)
    {
      theKey = inKey;
      val = inVal;
    }
    //ASSUMPTION: The << must be available for both the type of the
    //            value and the type of the key!
    void Print()
    {
      cout << "  Data: " << val << endl;
      cout << "   Key: " << theKey << endl;
    }
};
```

Using The Multiply Templated Class

```
int main()
{
  int i;
  DataWithKey< int, string > students[4];
  DataWithKey< char, float > gradePerc[4];

  students[0].setInfo(4819, "Homer");
  students[1].setInfo(9811, "Marge");
  students[2].setInfo(1624, "Bart");
  students[3].setInfo(3902, "Lisa");

  for (i = 0; i < 4; i++)
  {
    cout << "Student #: " << students[i].getKey() <<
         " Name: " << students[i].getVal() << endl;
  }

  gradePerc[0].setInfo('A', 88.5);
  gradePerc[1].setInfo('B', 75.0);
  gradePerc[2].setInfo('C', 64.5);
  gradePerc[3].setInfo('D', 56.25);
  for (i = 0; i < 4; i++)
  {
    cout << "Grade: " << gradePerc[i].getKey() <<
            " Min %age: " << gradePerc[i].getVal() << endl;
  }
  return 0;
}
```

```
Student #: 4819 Name: Homer
Student #: 9811 Name: Marge
Student #: 1624 Name: Bart
Student #: 3902 Name: Lisa
Grade: A Min %age: 88.5
Grade: B Min %age: 75
Grade: C Min %age: 64.5
Grade: D Min %age: 56.25
```

Template Compile Issues

- As of this lecture, many compilers are not handling templates in a good way.
- Problem:
  - Many compilers do not allow you to split your templates into .h and .cpp files.
  - Instead, put your template interface in a .h file
  - Put your member function implementations in a ".inl" file
  - At the VERY bottom of your .h file (just before the #endif of your #include guards", use #include to include your .inl file.

its like a cpp file so you still need to #inlcude the things you need

```
#ifndef _FOO_TEMPLATE_H_
#define _FOO_TEMPLATE_H_
template< class T >
class FooClass
{
  void myFunc();
};
#include "FooClass.inl"
#endif // _FOO_TEMPLATE_H_
```

Contents of FooClass.h

```
template< class T >
void FooClass< T >::myFunc()
{
  …
}
```

Contents of FooClass.inl