Advanced Topics in Functions

Lecture 11

3.19 Unitary Scope Resolution Operator

- Unary scope resolution operator (`::`)
  - Access global variable if local variable has same name
  - Not needed if names are different
  - Use `::variable`
    - `y = ::x + 3;`
  - Good to avoid using same names for locals and globals

fig03_24.cpp (1 of 2)

```cpp
// Fig. 3.24: fig03_24.cpp
// Using the unary scope resolution operator.
#include <iostream>

using std::cout;
using std::endl;

#include <iomanip>

using std::setprecision;

// define global constant PI
const double PI = 3.14159265358979;

int main()
{
  // define local constant PI
  const float PI = static_cast< float >( ::PI );

  // display values of local and global PI constants
  cout << setprecision( 20 )
     << "  Local float value of PI = " << PI
     << "\nGlobal double value of PI = " << ::PI << endl;

  return 0;  // indicates successful termination
```

Access the global **`PI`** with **`::PI`**.

Cast the global **`PI`** to a **`float`** for the local **`PI`**. This example will show the difference between **`float`** and **`double`**.

fig03_24.cpp (2 of 2)

```cpp

} // end main
```

fig03_24.cpp output (1 of 1)

Borland C++ command-line compiler output:

Local float value of PI = 3.1415927410125732
Global double value of PI = 3.141592653589790007

Microsoft Visual C++ compiler output:

Local float value of PI = 3.1415927410125732
Global double value of PI = 3.14159265358979

3.20 Function Overloading

- Function overloading
  - Functions with same name and different parameters
  - Should perform similar tasks
    - I.e., function to square **int**s and function to square **float**s

```
int square( int x) {return x * x;}
float square(float x) { return x * x; }
```

- Overloaded functions distinguished by signature
  - Based on name and parameter types (order matters)
  - Name mangling
    - Encodes function identifier with parameters
  - Type-safe linkage
    - Ensures proper overloaded function called

fig03_25.cpp (1 of 2)

```
// Fig. 3.25: fig03_25.cpp
// Using overloaded functions.
#include <iostream>

using std::cout;
using std::endl;

// function square for int values
int square( int x )
{
   cout << "Called square with int argument: " << x << endl;
   return x * x;

} // end int version of function square

// function square for double values
double square( double y )
{
   cout << "Called square with double argument: " << y << endl;
   return y * y;

} // end double version of function square
```

Overloaded functions have the same name, but the different parameters distinguish them.

(2 of 2)

fig03_25.cpp output (1 of 1)

```
int main()
{
   int intResult = square( 7 );        // calls int version
   double doubleResult = square( 7.5 ); // calls double version

   cout << "\nThe square of integer 7
      << "\nThe square of double 7.5
      << endl;

   return 0;  // indicates successful termination

} // end main
```

The proper function is called based upon the argument (**`int`** or **`double`**).

```
Called square with int argument: 7
Called square with double argument: 7.5

The square of integer 7 is 49
The square of double 7.5 is 56.25
```


fig03_26.cpp (1 of 2)

```
// Fig. 3.26: fig03_26.cpp
// Name mangling.

// function square for int values
int square( int x )
{
   return x * x;
}

// function square for double values
double square( double y )
{
   return y * y;
}

// function that receives arguments of types
// int, float, char and int *
void nothing1( int a, float b, char c, int *d )
{
   // empty function body
}

```

(2 of 2)

fig03_26.cpp output (1 of 1)

```
// function that receives arguments of types
// char, int, float * and double *
char *nothing2( char a, int b, float *c, double *d )
{
   return 0;
}

int main()
{
   return 0;  // indicates successful termination

} // end main
```

```
_main
@nothing2$qcipfpd
@nothing1$qifcpi
@square$qd
@square$qi
```

Mangled names produced in assembly language.

`$q` separates the function name from its parameters. **`c`** is **`char`**, **`d`** is **`double`**, **`i`** is **`int`**, **`pf`** is a pointer to a **`float`**, etc.

3.21 Function Templates

- Compact way to make overloaded functions
  - Generate separate function for different data types
- Format
  - Begin with keyword **`template`**
  - Formal type parameters in brackets **`<>`**
    - Every type parameter preceded by **`typename`** or **`class`** (synonyms)
    - Placeholders for built-in types (i.e., **`int`**) or user-defined types
    - Specify arguments types, return types, declare variables
  - Function definition like normal, except formal types used

3.21 Function Templates

- Example

```
template < class T > // or template< typename T >
T square( T value1 )
{
   return value1 * value1;
}
```

  - **T** is a formal type, used as parameter type
    - Above function returns variable of same type as parameter
  - In function call, T replaced by real type
    - If **int**, all **T**'s become **int**s

```
int x;
int y = square(x);
```


fig03_27.cpp (1 of 3)

```
// Fig. 3.27: fig03_27.cpp
// Using a function template.
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

// definition of function template maxi
template < class T >  // or template < typename T >
T maximum( T value1, T value2, T value3 )
{
  T max = value1;

  if ( value2 > max )
    max = value2;

  if ( value3 > max )
    max = value3;

  return max;

} // end function template maximum

```

Formal type parameter **T** placeholder for type of data to be tested by **maximum**.

maximum expects all parameters to be of the same type.

fig03_27.cpp (2 of 3)

```
int main()
{
   // demonstrate maximum with int values
   int int1, int2, int3;

   cout << "Input three integer values: ";
   cin >> int1 >> int2 >> int3;

   // invoke int version of maximum
   cout << "The maximum integer value is: "
      << maximum( int1, int2, int3 );

   // demonstrate maximum with double values
   double double1, double2, double3;

   cout << "\n\nInput three double values: ";
   cin >> double1 >> double2 >> double3;

   // invoke double version of maximum
   cout << "The maximum double value is: "
      << maximum( double1, double2, double3 );

```

`maximum` called with various data types.

fig03_27.cpp (3 of 3)

```
   // demonstrate maximum with char values
   char char1, char2, char3;

   cout << "\n\nInput three characters: ";
   cin >> char1 >> char2 >> char3;

   // invoke char version of maximum
   cout << "The maximum character value is: "
      << maximum( char1, char2, char3 )
      << endl;

   return 0;  // indicates successful termination

} // end main
```

fig03_27.cpp output (1 of 1)

```
Input three integer values: 1 2 3
The maximum integer value is: 3

Input three double values: 3.3 2.2 1.1
The maximum double value is: 3.3

Input three characters: A C B
The maximum character value is: C
```


Final Review

- Comprehensive Exam
- You'll have 2 hours to work on it
- Topics
  - Classes
  - Writing to/Reading from Files
  - Pointers
  - Control Structures
    - Loops
    - Conditionals
- More Topics
  - Arithmetic, casting, and type precedence
  - Arrays – 1 and 2D
  - string class
  - Recursion