C/C++ Programming
Session 6

Dr Jim Lupo
LSU/CCT Computational Enablement
jalupo@cct.lsu.edu
# Concept Review

<table>
<thead>
<tr>
<th>int main ( ... )</th>
<th>Float precision</th>
<th>if / else if / else</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ ... };</td>
<td>Promotion</td>
<td>switch</td>
</tr>
<tr>
<td>{ statement block; ... };</td>
<td>Machine $\epsilon$</td>
<td>break / continue / goto</td>
</tr>
<tr>
<td>Integer types</td>
<td>Operators</td>
<td>nesting statements</td>
</tr>
<tr>
<td>Float types</td>
<td>Bool types</td>
<td>data structs</td>
</tr>
<tr>
<td>String constants</td>
<td>Association</td>
<td>pointers</td>
</tr>
<tr>
<td>Variables</td>
<td>Precedence</td>
<td></td>
</tr>
<tr>
<td>cin / cout</td>
<td>for Loop</td>
<td></td>
</tr>
<tr>
<td>$&lt;&lt;$ and $&gt;&gt;$</td>
<td>while Loop</td>
<td></td>
</tr>
<tr>
<td>Storage in memory</td>
<td>do Loop</td>
<td></td>
</tr>
</tbody>
</table>
A Function

Trigonometry provides the archetypical concept of a function:

\[ x = \sin(\text{angle}) \]

This function:

1. Takes a single \textit{argument} – the value of an angle.
2. Applies some process using it.
3. \textit{Returns} the answer, which is stored in \( x \).

Haven't we seen this already?
Function Syntax

type fname ( type arg1, ..., type argN ) {
    ... statement block ...;
    return val;
}

Like variables, you have to create a name and declare the type of the result value return.

You get to decide how many arguments you need and what type.

Just make sure you return an answer when you're done.
Breaking the Rule

This is not recommended, but legal and you will see it:

```c
fname ( type arg1, ..., type argN ) {
    ... statement block ...;
    return val;
}
```

If you leave off the function type, it will be assumed to be `int`. 
Simple Functions

1. float addf ( float x, float y ) {
   return ( x + y );
}

4. double subf ( double x, double y ) {
   return ( x - y );
}

8. float m = 1.0F, o;
9. double a = 42., c;
10.
11. o = addf( m, 2.0F ); // 3.0F
12. c = subf( a, 52.0 ); // -10.0;
The list of arguments provides variable names (dummy arguments) which are replaced with the value of the actual arguments in the function call:

- `x, y` ..... the dummy arguments.
- `m, 2.0F` .. the actual arguments.
- `1.0F` ..... the value given to x.
- `2.0F` ..... the value given to y.
Terminology

When you use a function you are said to be *calling a function*.

Preparing the arguments, calling the function, how the arguments are handled, and dealing with the return value is the *calling sequence* (i.e. *call-by-value, call-by-address, call-by-descriptor*).

A function which does not return a result by design is a *void* function, and should be declared as such.

A function which has no arguments should declare a *void* argument list (i.e. `main(void)`).

Since functions may call functions, the *call stack* is the sequence of calls taken to reach a particular point in the program.
#include <iostream>
using namespace std;

float addf ( float x, float y ) {
    float sum;
    sum = x + y;
    x = 15.0F;
    return ( sum );
}

int main ( void ) {
    float m = 1.0F, o;
    o = addf( m, 2.0F );
    cout << "addf returned " << o << endl;
    cout << "m is now " << m << endl;
    return ( 0 );
}

Program output: addf returned 3
m is now 1

(See: addf.c/c++)
What Happened?

Line 12 – value of \( m \) is set to 1.
Line 13 – the values 1.0F and 2.0F are set as arguments.
Line 6 – the dummy arguments in the function use the values as if:

\[
\begin{align*}
x &= 1.0F; \\
y &= 2.0F;
\end{align*}
\]

The value of variable \( m \) isn't affected!
Multiple Results?

Here's a simple function to convert degrees Farenheit to degrees Celsius:

```c
float f2c ( float fahrenheit ) {
    float celsius;
    celsius = (5.0F/9.0F) * (fahrenheit - 32.0F);
    return ( celsius );
}
```

Since degrees Kelvin is related to degrees Celsius, could one function provide a conversion to both Celsius and Kelvin?

Hint: Use pointers?
# Multiple Return Values

```
1. #include <iostream>
2. using namespace std;
3.
4. int f2ck ( float dF, float *dC, float *dK ) {
5.   int errcode = 1;
6.   *dC = (5.0F/9.0F) * ( dF - 32.0F );
7.   *dK = *dC + 273.15;
8.   // No temperatures below absolute 0!
9.   if ( *dK >= 0. ) errcode = 0;
10.  return ( errcode );
11. }
12.
13. int main ( void ) {
14.   float dFahrenheit, dCelsius, dKelvin;
15.   if ( f2ck ( dFahrenheit, &dCelsius, &dKelvin ) == 0 ) {
16.      cout << "Okay, " << dCelsius << " C; " << dKelvin << " K" << endl;
17.   } else {
18.      cout << "Bad! " << dFahrenheit << " F below absolute 0!" << endl;
19.   }
20.  return ( 0 );
21. }
```

(See: f2ck_ptr.c/c++)
How Does It Work?

Line 4 – the dummy arguments include pointers dC and dK. Remember that the value of a pointer is an address to a variable of the pointer's type.
Line 6 – assigns a value to the variable pointed to by dC.
Line 7 – assigns a value to the variable pointed to by dK.
Line15 – sets the addresses of dCelsius and dKelvin as the values to pass to the function.

Assigning a value to *dC changes the value of dCelsius.
Assigning a value to *dK changes the value of dKelvin.
```
1. using namespace std;
2.
3. int f2ck ( float dF, float *results ) {
4.     int errcode = 1;
5.     results[0] = (5.0F/9.0F) * ( dF - 32.0F );
6.     results[1] = results[0] + 273.15;
7.     // No temperatures below absolute 0!
8.     if ( results[1] >= 0. ) errcode = 0;
9.     return ( errcode );
10. }
11.
12. int main ( void ) {
13.     float dFahrenheit = 72.0F, converted[2];
14.     if ( f2ck ( 72.0F, converted ) == 0 ) {
15.         cout << "Okay, " << converted[0] << "C; "
16.             << converted[1] << "K" << endl;
17.     } else {
18.         cout << "Bad! " << dFahrenheit << "F below absolute 0!" << endl;
19.     }
20.     return ( 0 );
21. }
```

Line 14 – **converted** is equivalent to **&converted[0]**

(See: f2ck_array.c/c++)
Additional Variable Attributes

**scope** ..... when is a variable name visible to the compiler.
**lifetime** .. when does a variable's storage exist at runtime.

As soon as you start using functions, the single most critical question becomes: where should I define variables, and how do they behave? Fact is, storage in memory is directly related to where in the program the variable appears.
#include <iostream>
using namespace std;

float addf ( float x, float y ) {
    float sum;
    sum += x + y;
    x = 15.0F;
    return ( sum );
}

int main ( void ) {
    float m = 1.0F, o;
    o = addf( m, 2.0F );
    o = addf( m, 2.0F );
    cout << "addf returned " << o << endl;
    cout << "m is now " << m << endl;
    return ( 0 );
}
1. `#include <iostream>
2. using namespace std;
3. 
4. `float addf ( float x, float y ) {
5.     float sum = 0.0F;
6.     sum += x + y;
7.     x = 15.0F;
8.     return ( sum );
9. }`

10.
11. `int main ( void ) {
12.     float m = 1.0F, o;
13.     o = addf( m, 2.0F );
14.     o = addf( m, 2.0F );
15.     cout << "addf returned " << o << endl;
16.     cout << "m is now " << m << endl;
17.     return ( 0 );
18. }
```

This is a safe example!

Program output will be:  `addf returned 3 m is now 1`
Lifetime

So, the *lifetime* of an automatic variable is typically the time it takes for a function to execute.

Let's see if we can take on the concept of *scope*. 
File Scope

1. `#include <iostream>
2. using namespace std;
3. float sum;
4. 
5. float addf ( float x, float y ) {
   6.    sum += x + y;
   7.    return ( x + y );
   8. }
9. 
10. int main ( void ) {
11.    float o = 0.0F;
12.    cout << "sum is " << sum << "; o is " << o << endl;
13.    o = addf( 1.0F, 2.0F );
14.    cout << "sum is " << sum << "; o is " << o << endl;
15.    o = addf( 100.0F, 2000.0F );
16.    cout << "sum is " << sum << "; o is " << o << endl;
17.    return ( 0 );
18. }

Program output:

sum is 0; o is 0
sum is 3; o is 3
sum is 2103; o is 2100
Unit Scope

1. #include <iostream>
2. using namespace std;
3. float sum;
4. 
5. float addf ( float x, float y ) {
6.     float sum;
7.     sum += x + y;
8.     return ( x + y );
9. }
10. 
11. int main ( void ) {
12.     float o = 0.0F;
13.     cout << "sum is " << sum << " ; o is " << o << endl;
14.     o = addf( 1.0F, 2.0F );
15.     cout << "sum is " << sum << " ; o is " << o << endl;
16.     o = addf( 100.0F, 2000.0F );
17.     cout << "sum is " << sum << " ; o is " << o << endl;
18.     return ( 0 );
19. }

Program output:

sum is 0; o is 0
sum is 0; o is 3
sum is 0; o is 2100
Differences

- File scope variables are visible anywhere in the file.
- Unit scope variables take precedence and are visible only in the program unit that defines them.
- Unit scope variables have limited lifetimes.
- File scope variables persist through the program execution.

There is one other scope, called global, which creates variables that are visible even across files.
Scope Applies to Any Identifier

```c
int bar ( int a, int b ) {
    ...
}

int foo ( int a, int b ) {
    int bar ( int a, int b ) {
        ...
    }
    int x = bar ( 5, 11 );
}
```

The function `bar inside` of `foo` masks the function `bar outside` of `foo`. 
Value Persistence Between Calls?

Automatic variables disappear when they go out of scope.

What if you want a function to accumulate a value?

Can tell the compiler to create a static variable!

Such a variable persists between function calls!
```
1. #include <iostream>
2. using namespace std;
3.
4. float addf ( float x, float y ) {
5.    static float sum;
6.    sum += x + y;
7.    return ( sum );
8. }
9.
10. int main ( void ) {
11.    cout << "addf returned " << addf( 1.0F, 2.0F ) << endl;
12.    cout << "addf returned " << addf( 1.0F, 2.0F ) << endl;
13.    cout << "addf returned " << addf( 1.0F, 2.0F ) << endl;
14.    return ( 0 );
15. }
```

Program output:

```
addf returned 3
addf returned 6
addf returned 9
```

(See: static.c/c++)
Function Prototypes

With variables, we could choose to declare their types, but allocate storage with definitions. This allowed us to use header files.

Prototypes allow us to do much the same with functions. We've been working with function definitions so far. A function prototype looks like a function definition without a statement block.
Function Prototypes

With variables, we could choose to declare their types, but allocate storage with definitions. This allowed us to use header files.

Prototypes allow us to do much the same with functions. We've been working with function definitions so far. A function prototype looks like a function definition without a statement block.

```
float addf ( float, float );
```

Note: no dummy arg names or statement block – just end with ;.
A Best Practice

May people adopt a *one function per file* rule.

**addf.h:**

```c
float addf ( float, float );
```

**addf.C:**

```c
#include "addf.h"
float addf ( float x, float y ) {
    sum += x + y;
    return ( x + y );
}
```

**program.C:**

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

int main ( void ) {
    float m = 1.0F, o;
    o = addf( m, 2.0F );
    ... 
}
```
Compiling Multiple Files

The compile command changes a little:

```
icc -o addf addf.C program.C
```

The idea is to help you organize a large project and make it more or less obvious where to start looking for problems or to make changes. 1 hundred 1,000 line files are easier to manage than 1 100,000 line file.

Will revisit this concept later, and will have to get serious about specifying where a variable is actually defined! It can only be done once!
Advanced Topic

What if you wish the function to change the value of the arguments passed it?

Method 1: use pointers (legal, but dangerous):

```c
void foo ( int *x, int *y ) { *x += 5; *y /= 2; }
int a = 100, b = 100;
foo ( &a, &b );
```

The *values* passed are the *addresses* of the arguments. This is available in C and C++. 
Method 1 Dangers

Typo may be legal, but not do what you want:

```c
void foo ( int *x, int *y ) { x += 5; *y /= 2; } 
int a = 100, b = 100;
foo ( &a, &b );
```

May attempt to access memory illegally, or worse, some arbitrary legal location overwriting valid data.

```c
void foo ( int *x, int *y ) { *x += 5; *y /= 2; } 
int a = 100, b = 100;
foo ( 100, &b );
```
Method 2: Call By Reference

```c
void foo ( int& x, int& y ) { x += 5; y /= 2; }
int a = 100, b = 100;
foo ( a, b );
```

You can think of this as implicit call-by-address, but you do not use pointers. (This is the Fortran default method, by the way.)

Same memory movement efficiencies as pointers.

This is available only in C++. 
Method 2 Safety Features

```c
void foo ( int& x, int& y ) { x += 5; y /= 2; }
int a = 100, b = 100;
foo ( a, b );
```

No confusion between pointer address and reference value, but one can not call-by-value without a lot of pain.

Compiler will give warning if variable names aren't present in a function call (i.e. `foo(1,b)` will cause error).
Preventing Modification

One reason to use call-by-reference (or pointers) is to prevent excessive copying of memory. What if the function needs access to the values, but is not supposed to change them?

Tell the compiler the argument is a constant!

```c
void foo ( int& x, const int& y ) { x += y - 5; } 
int a = 100, b = 100;  
foo ( a, b );
```

Any attempt to change `y` will generate an error.
Comments?

Questions?