C/C++ Programming
Session 3

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

http://reu.cct.lsu.edu/documents/C_Course/index.php
# Concept Review

<table>
<thead>
<tr>
<th>Code Snippet</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int main ( ... ) { ... }</td>
<td>Float precision</td>
</tr>
<tr>
<td>statement;</td>
<td>Promotion</td>
</tr>
<tr>
<td>{ statement block }</td>
<td>Machine &amp;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer types</td>
<td></td>
</tr>
<tr>
<td>Float types</td>
<td></td>
</tr>
<tr>
<td>String constants</td>
<td></td>
</tr>
<tr>
<td>Variables</td>
<td></td>
</tr>
<tr>
<td>cin / cout</td>
<td></td>
</tr>
<tr>
<td>&lt;&lt; and &gt;&gt;</td>
<td></td>
</tr>
<tr>
<td>Storage in memory</td>
<td></td>
</tr>
</tbody>
</table>
Choosing Variable Types

Imagine working on an inventory program. An initial review of requirements suggests the following must be accounted for, so some variables are defined. Are they correct?

<table>
<thead>
<tr>
<th>Description</th>
<th>Legal Values</th>
<th>Type &amp; Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>vendor part number</td>
<td>&lt; 4,000,000,000</td>
<td>int vendor_pn;</td>
</tr>
<tr>
<td>local part number</td>
<td>&lt; 10,000,000</td>
<td>int local_pn;</td>
</tr>
<tr>
<td>warehouse</td>
<td>A – L</td>
<td>char warehouse;</td>
</tr>
<tr>
<td>row</td>
<td>1 - 200</td>
<td>int row</td>
</tr>
<tr>
<td>shelf</td>
<td>A – Z</td>
<td>char shelf;</td>
</tr>
<tr>
<td>bin</td>
<td>1 - 32</td>
<td>int bin;</td>
</tr>
<tr>
<td>quantity</td>
<td>&lt; 32,000</td>
<td>int count;</td>
</tr>
<tr>
<td>unit price</td>
<td>&lt; $10,000.00</td>
<td>double price;</td>
</tr>
</tbody>
</table>
A Better Solution

<table>
<thead>
<tr>
<th>Description</th>
<th>Legal Values</th>
<th>Type &amp; Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>vendor part number</td>
<td>&lt; 4,000,000,000</td>
<td>long vendor_pn;</td>
</tr>
<tr>
<td>local part number</td>
<td>&lt; 10,000,000</td>
<td>int local_pn;</td>
</tr>
<tr>
<td>warehouse</td>
<td>A – L</td>
<td>char warehouse;</td>
</tr>
<tr>
<td>row</td>
<td>1 - 200</td>
<td>unsigned char row</td>
</tr>
<tr>
<td>shelf</td>
<td>A – Z</td>
<td>char shelf;</td>
</tr>
<tr>
<td>bin</td>
<td>1 - 32</td>
<td>unsigned char bin;</td>
</tr>
<tr>
<td>quantity</td>
<td>&lt; 32,000</td>
<td>short count;</td>
</tr>
<tr>
<td>unit price</td>
<td>&lt; $10,000.00</td>
<td>float price;</td>
</tr>
</tbody>
</table>

Future changes? Do we really have to save the memory?
Operators

An *operator* defines a particular arithmetic procedure applied to some set of indicated *operands*. In C/C++ there are three types of operator:

- unary .... requires a single operand.
- binary ... requires two operands.
- ternary .. requires three operands.

Some examples will make it less scary than the definitions might imply.
Binary Operators

+ . . Addition
- . . Subtraction
* . . Multiplication
/ . . Division
% . . Modulus or remainder

No big surprises here, they work just like they do in algebra.

Well, maybe just one insidious caveat: / and % applied to negative numbers are allowed to produce *machine dependent results.*
Association and Precedence

Compound statements are interpreted by applying the rules of association and precedence. The arithmetic operators associate left to right, and higher precedence operators are applied first.

For instance: \( a \ast b + c \)

is evaluated as: \((a \ast b) + c\)

not as: \( a \ast (b + c)\)

Group an expression using ( ) to be explicit in grouping operands.
Relational & Logical Operators

== . . equal to
!= . . not equal to
> . . . greater than
>= . . greater than or equal to
< . . . less than
<= . . less than or equal to
&& . . logical AND.
|| . . logical OR
! . . . logical NOT

The results are a boolean value of true or false.
bool Type

C++ has a type `bool` (as in Boolean logic) which may only store the value `true` or `false` (defined named constants):

1. `bool answer, x, y;`
2. `x = false;`
3. `y = true;`
4. `answer = x || y; // yields true.`
5. `x = 42.0 > 96.0;`
6. `y = 13.0 > 12.999;`
7. `answer = x && y; // yields false.`
bool Values

By convention, a value of 0 is treated as `false`, and any non-zero value is treated as `true`.

The actual value used for the `language defined variables true and false` are compiler dependent!
Examples

1. bool result;
2. int x, y;
3. 
4. x = 42;
5. y = 7;
6. result = (x >= 42) && (y < 8);
7. result = (y == 7) || (x == 15);
8. result = (y*6 == x) && ! (y == 5);
9. result = (y = x - 41);

Line 9 is a sneaky, legal, C trick.
Output \texttt{true/false} as Strings

1. \#include \texttt{<iostream>}
2. \#include \texttt{<iomanip>}
3. using namespace std;
4. int main ( void ) {
5. \hspace{1em} bool result;
6. \hspace{1em} int x, y;
7. \hspace{1em} x = 42;
8. \hspace{1em} y = 7;
9. \hspace{1em} result = ( x \geq 42 ) && ( y < 8 );
10. \hspace{1em} cout.\texttt{setf( ios::boolalpha )};
11. \hspace{1em} cout << "( x > 42 ) && ( y < 8 ) is "
12. \hspace{1em} \hspace{1em} << result << \texttt{endl};
13. \hspace{1em} return ( 0 );
14. }

Produces: ( x > 42 ) && ( y < 8 ) is true
Unary Increment/Decrement

Prefix form:  

++i  /  --i

The value of $i$ is incremented / decremented by 1, then it is used.

Postfix form:  

i++  /  i--

The value of $i$ is used, then the value of $i$ is incremented / decremented by 1.

Either form may only be applied to a variable name, since a value must be changed and stored somewhere.
Example

1. int i, j, k;
2.
3. i = 10;
4. j = i--; // After: j is 10, i is 9.
5. i = 10;
6. k = ++i; // After: k is 11, i is 11.
Bitwise Binary Operators

May only be applied to integer variables:

\texttt{&} \ldots \text{ bitwise AND}
\texttt{|} \ldots \text{ bitwise inclusive OR}
\texttt{^} \ldots \text{ bitwise exclusive OR (XOR)}
\texttt{<<} \ldots \text{ left shift, 0 fill.}
\texttt{>>} \ldots \text{ right shift, 0 fill.}

These \textit{twiddle the bits} and produce new integer values, thus you must think in binary values.
Bitwise Unary Operator

\[ \sim \ldots 1\text{'s complement} \]

This has the effect of changing every 1-bit in a value to 0, and every 0-bit to a 1.

\[ \sim 14 \ (0b00001110) \text{ becomes } 241 \ (0b11110001) \]

This can yield surprising results if you don't keep in mind signed and unsigned data types!
Examples

```c
1. unsigned short i, j, k;
2. i = 7; // b'00000111
3. j = 4; // b'00000100
4.
5. k = i & j; // 4   (b'00000100)
6. k = i ^ j; // 3   (b'00000011)
7. k = i << 3; // 56  (b'00111000)
8. k = j << j; // 64  (b'01000000)
9. k = i >> 2; // 1   (b'00000001)
```

The bit manipulation operators apply Boolean logic to each pair of bits.
Context is Important

<<, >> . . are stream operators in an I/O context.

<<, >> . . are bit operators in an arithmetic context.

This actually illustrates the concept of operator overloading, which means that an operator symbol can be defined to do different things in different contexts. It is a feature of *object oriented programming*. 
Assignment Operators

Reassignment statements, such as:

\[ i = i + 5 \]

appear so often in coding, that C/C++ provide assignment operators to abbreviate this to:

\[ i += 5 \]

There is a form for each of the binary operators:

\[ -= += *= /= %= &= ^= |= <<= >>= \]
Examples

1. int x, y, z;
2. x = 42;
3. y = 3;
4. z = 10;
5. y += x + z; // 55 42 10
6. y *= x + z; // 2860 42 10
7. z -= 3; // 2860 42 7
Expressions and Sub-expressions

From a legal statement, we can identify expressions and sub-expressions. For instance, consider this full statement:

\[ z = a \times (b + c); \]

The right-hand side (RHS) is an arithmetic expression because it returns an arithmetic result. The contents inside the ( ) form a sub-expression, which is also arithmetic.
An Assignment Expression

An assignment operation can be used as an expression, the value of which is the result of the expression:

\[ x = 45.0; \]
\[ y = x; \]

could be written as:

\[ y = ( x = 45.0 ); // or: \ y = x = 45.0; \]
Ternary Operator

Think of assigning a value based on the result of some test. The test would be some question that yields true or false, and you'd like to use one value of if the result is true, and a different value if it is false. The `?` operator will do that:

```
result = exp_test ? exp_true : exp_false;
```

Requires 3 complete expressions:
- `exp_test` ... a logical (true/false) expression.
- `exp_true` ... a value expression used if `exp_test` is true.
- `exp_false` ... a value expression used if `exp_test` is false.

Type of `exp_true` and `exp_false` should match type of `result`. 
Ternary Example

1. `int state, color, go, stop;`
2. `int red, green;`
3. `red = -1;`
4. `green = 1;`
5. `go = 42;`
6. `stop = -42;`
7. `state = 0;`
8. `color = ( state == go ) ? green : red;`
9. `cout << color << endl;`
10. `// Basically defaults color to RED!`
Self-Check 1

1. `int i, j, k, l, m;`
2. `i = 5 % 2;`
3. `j = 5 / 2;`
4. `k = 5 * 2;`
5. `l = 5 + 2;`
6. `m = 5 - 2;`
Self-Check 2

1. `bool i, j, k;`
2. `i = 1 == 5 % 2;`
3. `j = 1 == 5 / 2;`
4. `k = 1 != 5 / 2;`
Self-Check 3

1. int i;
2. bool j, k, l, m;
3. i = 5;
4. j = 3 == ++i / 2;
5. k = 3 == i++ / 2;
6. l = 1 == i++ % 2;
7. m = ++i % 2 == 0;
Self-Check 4

What is the type and value returned by these expressions?

A) \( 5 + 6 \times \frac{7}{8} \)
B) \( 5.0 + 6.0 \times \frac{7.0}{8} \)
C) \( 5 + 6 \times \frac{7}{8} \).
Implicit Promotion

In a binary expression which contains mixed types, the compiler will promote the lower type to match the higher type before performing the operation. For example:

```c
int bar, i, j;
double foo, x, y;
float fie, a, b;
foo = 6 / a;  // 6/a → 6.0F/a → float → double foo
j = 6 / 3.0;  // 6/3.0 → 6.0D/3.0D → 2.0D → int 2
```

An optimizing compiler may do this in fewer steps:

```c
foo = 6 / a;  // 6/a → 6.0D/(double a) → double foo
```
Take Aways

- Unary, Binary, Ternary Operators
- Precedence
- Distinction between *statement* and *expression*
- Operator behavior determined by context.
- Impact on implicit conversions

Program examples on available on the REU web site.