C/C++ Programming
Session 4

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

<table>
<thead>
<tr>
<th>int main ( ... )</th>
<th>Float precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ ... }</td>
<td>Promotion</td>
</tr>
<tr>
<td>statement;</td>
<td>Machine $\epsilon$</td>
</tr>
<tr>
<td>{ statement block; ... };</td>
<td></td>
</tr>
<tr>
<td>Integer types</td>
<td>Operators</td>
</tr>
<tr>
<td>Float types</td>
<td>Bool types</td>
</tr>
<tr>
<td>String constants</td>
<td>Association</td>
</tr>
<tr>
<td>Variables</td>
<td>Precedence</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>
Mix and Match

Which class can be applied to which operation types?

<table>
<thead>
<tr>
<th>Class</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unary</td>
<td>arithmetic</td>
</tr>
<tr>
<td>Binary</td>
<td>relational</td>
</tr>
<tr>
<td>Ternary</td>
<td>bitwise</td>
</tr>
<tr>
<td></td>
<td>assignment</td>
</tr>
<tr>
<td></td>
<td>conditional</td>
</tr>
</tbody>
</table>
What's Wrong With This?

1. bool result;
2. float x;
3. x = 42.0;
4. 
5. result = ( x == 45.0 );
6. 
7. result = ( x = 45.0 );
Machine & Calculation

Flash back to Session 2: Is there a better way?

1. #include <iostream>
2. #include <iomanip>
3. using namespace std;
4. int main ( void ) {
5.    float epsilon;
6.    epsilon = 1.0F;
7.    cout << setprecision(20) << 1.0F - epsilon/2.0F << " : " << epsilon << endl;
8.    
9.    epsilon = epsilon / 2.0F;
10.   ...
11.   ... repeat lines 7 & 9 till no change shown by cout
12.   }
Programming Requirements

1. Some way to repeat statements automatically, maybe with different variable values.
2. Some way to stop the program at the answer.

Might it be nice to *loop* over some statement block?

How about use relational operators to test conditions, and maybe stop the loop when the answer is found?

Yes! Its possible! All courtesy of the magic of control structures!
The **for** Loop

Handy if we can count how many times we want to do something (and other things):

```c
for ( <initExpr>; // Initializer expression
    <testExpr>; // Conditional expression
    <incrExpr> ) // Increment expression
{
    statement block;
}
```
The **for** Semantics

1. Evaluate `init_exp` – sets a counting variable.  
2. Evaluate `test_exp` – if `false`, jump to Step 6; if `true`, jump to Step 3.  
3. Evaluate the statement block.  
4. Evaluate `incr_expr` – change counting variable.  
5. Jump to Step 2.  
for Initialization Expression

Initialize one or more variables, comma separated.

```c
int i, j;
for ( i = 0; ... ; ... ) {...}
for ( i = 0, j = 10; ... ) {...}
```

But it is an expression, so not limited to just initializing.
for Test Expression

Expression whose result is treated as true or false.

```c
int i, j;

for ( i = 0; i < 10; ... ) { ..sb.. }
for ( i = -10, j = 0;
    j < 42; ... ) { ..sb.. }
```

If test expression is **true**, the statement block (**sb**) is executed.
for Increment Expression

After evaluating statement block, evaluate increment expression, then re-evaluate the test expression.

```c
int i, j;
for ( i = 0; i < 10; i++ ) {...}
for ( i = -10, j = 0;
    i < 10 && j < 42;
    i++, j += 3 ) {...}
```

The whole process repeats until the test expression evaluates to false.
Full Example

1. int i;
2. float tf, tc;
3. for ( i = -10, tf = 300.0;
4.     i < 10; tf += 0.5, i++ ) {
5.     tc = (5.0/9.0) * ( tf - 32.0 );
6.     cout << tf << " F is "
7.       << tc << " C" << endl;
8.     cout << "Counter is now " << i
9.       << endl;
10. }
# Match Task with ID

<table>
<thead>
<tr>
<th>Task</th>
<th>Task Desired</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Count by 5s to 80</td>
</tr>
<tr>
<td>B</td>
<td>Decrement to -10</td>
</tr>
<tr>
<td>C</td>
<td>Step by 3 to 1000</td>
</tr>
<tr>
<td>D</td>
<td>Steps by 1 and 5</td>
</tr>
<tr>
<td>E</td>
<td>Never stop</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>Choice of for Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( j = 1; j &lt;= 1000; j += 3 )</td>
</tr>
<tr>
<td>2</td>
<td>( i = 0; i &lt; 100; i++, j += 5 )</td>
</tr>
<tr>
<td>3</td>
<td>( k = 0; k &lt;= 80; k +=5 )</td>
</tr>
<tr>
<td>4</td>
<td>( i = 10; i &gt; 0; i++ )</td>
</tr>
<tr>
<td>5</td>
<td>( j = 3000; j &gt;= -10; j -= 10 )</td>
</tr>
<tr>
<td>6</td>
<td>( k = 0; k &lt; 80; k += 5 )</td>
</tr>
</tbody>
</table>
Use a `for` Loop for Machine ε?

```
1. #include <iostream>
2. #include <iomanip>
3. using namespace std;
4. int main ( void ) {
5.     float epsilon = 1.0F;
6.     float diff = 0.0F;
7.     int i;
8.     for ( i = 0; i < ???; i++ ) {
9.         diff = 1.0F - epsilon/2.0F;
10.        cout << setprecision(20)
11.        << diff
12.        << " : " << epsilon << endl;
13.        epsilon = epsilon / 2.0F;
14.     }
15. }
```
Better Decide on a Test

Recall the desired condition was:

\[ 1 - \varepsilon \neq 1 \]
```cpp
1. #include <iostream>
2. #include <iomanip>
3. using namespace std;
4. int main ( void ) {
5.     float epsilon = 1.0F;
6.     diff = 0.0F;
7.     int i;
8.     for ( i = 0; diff != 1.0F;
9.         epsilon /= 2.0F, i++ ) {
10.        diff = 1.0F - epsilon;
11.        cout << setprecision(20) << diff
12.            << " : " << epsilon << endl;
13.     }
14.     cout << setprecision(20) << epsilon * 2.0F
15.        << "; " << i << " iterations." << endl;
16.     return ( 0 );
17. }
```
The **while** Loop

A common problem may be to stop processing when a desired condition is reached but the number of steps needed is uncomputable:

```c
while ( test_exp ) { ... }
```

1. The `test_exp` is evaluated.
   - If result is `false`, jump to line 4.
   - If result is `true`, jump to line 2.
2. Evaluate statement block.
4. Continue with next statement after block.
The **do** Loop

The **while** loop puts the conditional test at the beginning of the statement block. An alternative, the **do** loop, puts the test at the end:

```
  do {
      ...
  } while ( test_exp );
```

Note that the **do** statement block is *always* executed, while it might not be for the **while** loop.
The if Statement

We've been constructing descriptions using boolean logic: if something is true then do something. There is a way to program exactly this type of control. The simplest form:

```c
if ( test_exp ) { ... }
```

If `test_exp` evaluates to `true`, then the statement block is executed, otherwise it is ignored.
Simple Examples

```c
if ( hungry ) {
    ... define eating process
}
```

A different logic sense might be:

```c
if ( ! satiated ) {
    ... define eating process
}
```

In either case, eating is not performed unless hunger is present.
Alternate Action

A slightly more complete form allows an alternate action:

```c
if ( test_exp ) {
    ... do if true statement block ...
} else {
    ... do if false statement block ...
}
```

One or the other statement block will be executed.
if/else Example

1. int red=0, green=1, blue=42, yellow=18;
2. int color;
3. { ...
4.   do some stuff to change color
5.   ...
6. } if (color == blue) {
7.     cout << "Why so sad?" << endl;
8. } else {
9.     cout << "Is " << color
10.     << " your happy color?" << endl;
11. }
Multiple Conditions

Since “if ( test_expr ) {...}” is a valid statement, chain conditions to make a complex test:

1. if ( test_expr1 ) {
2.     ...
3. } else if ( test_expr2 ) {
4.     ...
5. } else if ( test_expr3 ) {
6.     ...
7. } ... {
8.     ...
9. } else {
10.     ...
11. }
Stop Light

1. int red=0, yellow=1, green=2 }
2. int color;
3. { ... do something to change color ... }
4. if ( color == red ) {
5.   cout << "Hit the brakes!" << endl;
6. else if ( color == yellow ) {
7.   cout << "Hit the gas!" << endl;
8. } else {
9.   cout << "Just be cool!" << endl;
10. }

Note: the test expressions need not be related.
If they are, we can simplify things!
The `switch` Statement

1. `switch ( val_expr ) {`
2. `    case val1:`
3. `    ... ;`
4. `    break;`
5. `    case val2:`
6. `    ... ;`
7. `    break;`
8. `    case val3:`
9. `    ... ;`
10. `    case val4:`
11. `    ... ;`
12. `    break;`
13. `    default:`
14. `    ... ;`
15. `}`
Deconstructing `switch`: 1

1. `switch ( val_expr ) {`
2. `    case val1:`
3. `        ... ;`
4. `    break;`
5. `    case val2:`
6. `        ... ;`

`val_expr` must be an integer or char value.

`valn's` must be integer or char constants
Deconstructing `switch`: 2

6. ... ;
7. break;
8. case `val3`:
9. ... ;
10. case `val4`:
11. ... ;
12. break;

`break` marks end of statements for a `case`.

If missing, fall through to next `case` statements.

One way to think of this: each `case` marks where execution begins in the statement block; and the first `break` encountered marks where execution ends.
Deconstructing `switch`: 3

```c
11. ... ;
12. break;
13. default:
14. ... ;
15. }
```

`default` is a catch-all value. Every possible value allowed by `val_expr` must be accounted for by `case` entries. If the value is not found, and no `default` is specified, a program error will occur!
**switch Examples**

```c
int red=0, green=1, blue=2;
int color, year;

switch ( color ) {
    case red: ... ; break;
    case green: ... ; break;
}

switch ( year ) {
    case 1999: ... ; break;
    case 2012: ... ; break;
    case 1950: ... ; break;
    default: ...... }
```
#include <iostream>
#include <iomanip>
using namespace std;

int main ( void ) {
    float epsilon = 1.0F;
    float diff = 0.0;
    int i = 0;

    do {
        epsilon /= 2.0F;
        diff = 1.0F - epsilon;
        cout << setprecision(20) << diff
             << " : " << epsilon << endl;
        i++;
    } while ( diff != 1.0F );

    cout << setprecision(20) << epsilon * 2.0F
         << ";" << i << " iterations." << endl;
    return ( 0 );
}
Self-Assessment

List some pros/cons for choosing between a switch statement or an if/elseif/else construct for:

1. Bibliographic data
2. River flood stages
3. CO (carbon monoxide) warning level
4. Engine RPM red line
5. Aircraft stall warning
6. New car purchase
Comments?

Questions?