Outline

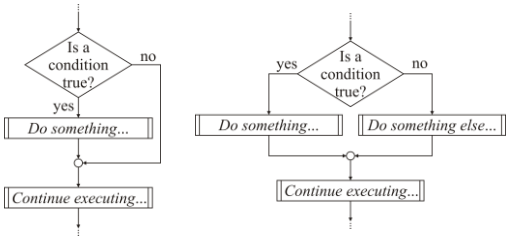
- In this lesson, we will:
  - Describe the need for executing code conditionally
  - Describe the flow chart and emphasize the purpose of flow charts
  - Describe the conditional statement
    - The absolute-value and max functions
  - Look at multiple conditional statements
    - Clipping and the tent function
  - Look at a simplification if there is no code to run if the statement is false
    - The sinc function
  - Finally, concluding with a simulation of the operational amplifier

Conditional statements

- In programming, we can *conditionally* execute code if some condition—a Boolean-valued statement—is satisfied (i.e., `true`)

Conditional statements

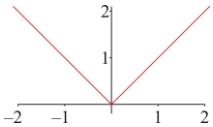
- There are two approaches in computer science to conditional statements:

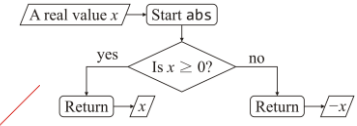


Conditional statements

- Many algorithms and mathematical formulas require that some condition be checked, and the code that is executed may differ based on the outcome of the condition
  - Consider the absolute value function:

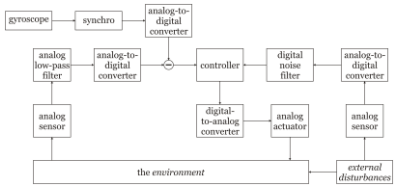
$$|x| \overset{\text{def}}{=} \begin{cases} x & x \geq 0 \\ -x & x < 0 \end{cases}$$





Block diagrams and flow charts

- For those programmers who are wondering
  "Why are we bothering with flow charts? Let's just do this now..."
- Flow charts and block diagrams are core to engineering:
  - Solutions for larger systems are broken up into steps, which step can be solved individually:



Block diagrams and flow charts

- At the time of writing this, this author had to come up with an algorithm to find the extreme values of a function correct to the highest possible floating-point precision
- While not so formally, the problem was broken into:
  - Appropriate sampling
  - Extrema detection
  - Application of quadratic optimization to localize each extrema
  - Application of a golden-mean search to further localize each extrema
  - Return the largest in absolute value

Conditional statements

- In order to execute code only if some condition is satisfied or not, we use a conditional statement:

```
if ( Boolean-valued condition ) {
    // The consequent block or body of statements
    //  - to be executed if the condition is true
} else {
    // The alternative block or body of statements
    //  - to be executed if the condition is false
}
```

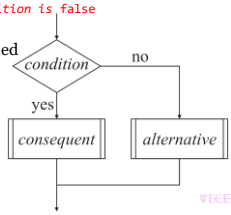
- Even though a conditional statement may have many statements within it, the entire structure is referred to as a conditional statement

Conditional statements

- We represent the condition with a diamond:

```
if ( Boolean-valued condition ) {
    The consequent block of statements
     - to be executed if the condition is true
} else {
    The alternative block of statements
     - to be executed if the condition is false
}
```

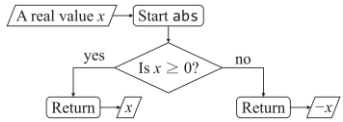
- The alternative consequences are labeled
  - The possibility of multiple statements are represented with additional bars



The absolute-value function

- Going back to the absolute-value function, the condition is "Is $x \geq 0$?"

```
double abs( double x ) {
    if ( x >= 0.0 ) {
        return  x;
    } else {
        return -x;
    }
}
```

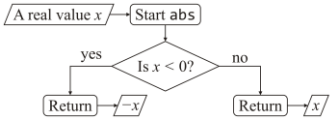


Using complementary operators

- If we use the complementary comparison operator, the consequent and alternative bodies are swapped:

```
double abs( double x ) {
    if ( x < 0.0 ) {
        return -x;
    } else {
        return  x;
    }
}
```

```
double abs( double x ) {
    if ( x >= 0 ) {
        return  x;
    } else {
        return -x;
    }
}
```



Unit step function

- The unit step function is a function defined as:

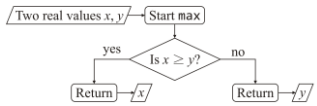
$$u(x) \stackrel{\text{def}}{=} \begin{cases} 1 & x \geq 0 \\ 0 & x < 0 \end{cases}$$

- Implement this function with the identifier `unit_step` and test it in main by printing out the results for three values

The max function

- As a second example, the maximum of two values is also based on a simple condition:

$$\max(x,y) \overset{\text{def}}{=} \begin{cases} x & x \geq y \\ y & x < y \end{cases}$$




```
double max( double x, double y ) {
    if ( x >= y ) {
        return x;
    } else {
        return y;
    }
}
```


Maximum of three

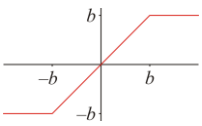
- Implement a function finding the maximum of three values:

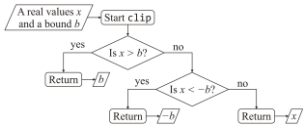
```
double max( double x, double y, double z );
```

- Follow these steps:
  - State your solution in English
  - Draw a flow chart
  - Write some tests
  - Write the function definition
- What are the appropriate tests?
  - The easiest is to try all permutations of `1.0`, `2.0` and `3.0`, the maximum of which in all cases should always be `3.0`

Clipping signals

- In engineering, signals (values) often cannot exceed certain bounds
  - If a signal x is greater in absolute value than some bound b, the bound is returned
  - If $x > b$, return b;
  - Otherwise:
    - If $x < -b$, return $-b$,
    - Otherwise, return x

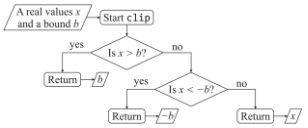




Clipping signals

- Implementing this, we have:

```
double clip( double x, double b ) {
    if ( x > b ) {
        return b;
    } else {
        if ( x < -b ) {
            return -b;
        } else {
            return x;
        }
    }
}
```



Once a return statement is executed, no subsequent statements in the function are executed

Clipping signals

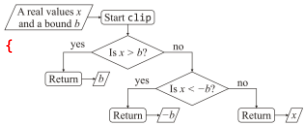
- If a conditional statement appears within either block of another conditional statement, we say that the statements are "nested"

```
double clip( double x, double b ) {
    if ( x > b ) {
        return b;
    } else {
        if ( x < -b ) {
            return -b;
        } else {
            return x;
        }
    }
}
```


Clipping signals

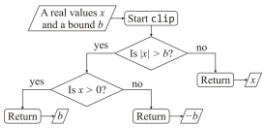
- If the **alternative block** is itself a conditional statement, we can *beautify* the code making it easier to read:
  - This is called an *else-if* statement within the conditional statement

```
double clip( double x, double b ) {
    if ( x > b ) {
        return  b;
    } else if ( x < -b ) {
        return -b;
    } else {
        return  x;
    }
}
```



Clipping signals

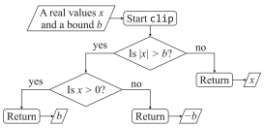
- Most problems have multiple solutions:
  - If $|x| > b$,
    - If $x > 0$, return b,
    - Otherwise, return $-b$;
  - Otherwise, return x.



Clipping signals

- Implementing alternative version, we have:

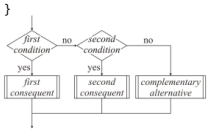
```
double clip( double x, double b ) {
    if ( abs( x ) > b ) {
        if ( x > 0 ) {
            return  b;
        } else {
            return -b;
        }
    } else {
        return x;
    }
}
```



Cascading conditional statements

- Such a sequence of if—else-if—⋯ statements is referred to as ***cascading*** *conditional statements*

```
double clip( double x, double b ) {
    if ( x > b ) {
        return  b;
    } else if ( x < -b ) {
        return -b;
    } else {
        return  x;
    }
}
```



Martin Püschel, *Song Khon Waterfall*

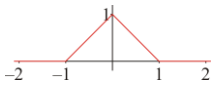
The *tent* function

- A *tent function* is defined as:

$$\text{tent}(x) \stackrel{\text{def}}{=} \begin{cases} 0 & x \le -1 \\ x+1 & -1 < x \le 0 \\ 1-x & 0 < x \le 1 \\ 0 & x > 1 \end{cases}$$

```
double tent( double x ) {
    if ( x <= -1.0 ) {
        return  0.0;
    } else if ( x <= 0.0 ) {
        return x + 1.0;
    } else if ( x <= 1.0 ) {
        return 1.0 - x;
    } else {
        return  0.0;
    }
}
```

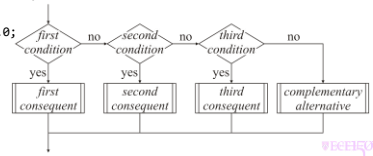
- What would the flow chart look like?



Cascading conditional statements

- Cascading conditional statements are represented by this flow chart
  - The alternative if complementary to all other consequences

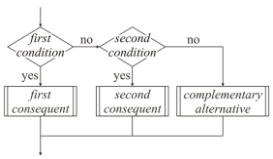
```
double tent( double x ) {
    if ( x <= -1.0 ) {
        return  0.0;
    } else if ( x <= 0.0 ) {
        return x + 1.0;
    } else if ( x <= 1.0 ) {
        return 1.0 - x;
    } else {
        return  0.0;
    }
}
```



Conditions are simply statements

- We can also use the `double abs(…)` function instead:

```
double tent( double x ) {
    if ( abs( x ) >= 1.0 ) {
        return  0.0;
    } else if ( x <= 0.0 ) {
        return x + 1.0;
    } else {
        return 1.0 - x;
    }
}
```



or even

```
double tent( double x ) {
    if ( abs( x ) >= 1.0 ) {
        return  0.0;
    } else {
        return 1.0 - abs( x );
    }
}
```

How not to use cascades

- Novice programmers sometimes want to emphasize the conditional checks:

```
if ( x >= 0 ) {
     // Do something...
} else if ( x < 0 ) {
     // Do something else...
}
```

```
if ( x == 0 ) {
    // Do something...
} else if ( x != 0 ) {
    // Do something else...
}
```

- Don't do this:
  - The last condition is complementary to the first
  - Experienced programmers reading this will be confused
    - They expect that there are some values of x that satisfy neither condition
  - Maintenance becomes more difficult

Assertions

- If mission-critical software requires you to be certain someone doesn't accidentally change the conditions, this is acceptable:

```
#include <cassert>

if ( x < -1 ) {
    // Do something...
} else if ( x <= 1 ) {
    // Do something else...
} else {
    assert ( x > 1 );
    // Do something else, yet again...
}
```

- assert
  - At runtime, this statement executes when the condition is false and prints to console

Common errors with cascades

- Consider this code:

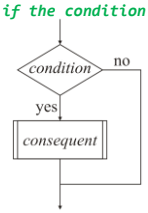
```
double square_wave( double x ) {
    if ( x <= -1.0 ) {
        return 0.0;
    } else if ( x < 0.0 ) {
        return -1.0;
    } else if ( x > 0.0 ) {
        return 1.0;
    } else if ( x >= 1.0 ) {
        return 0.0;
    } else {
        assert ( 0.0 == x );
        return 0.0;
    }
}
```

- What is the error in this cascade?

No alternative statement block

- If there is nothing to be done if the condition is false, this can be reduced to:

```
if ( Boolean-valued condition ) {
    The consequent block of statements
     - to be executed if the condition is true
}
```



The sinc function

- The *cardinal sine* or *sinc* function is defined as

$$\operatorname{sinc}(x) \overset{\text{def}}{=} \begin{cases} 1 & x = 0 \\ \dfrac{\sin(\pi x)}{\pi x} & x \neq 0 \end{cases}$$

- An implementation is:

```
double sinc( double x ) {
    // Deal with a special case when x = 0:
    if ( 0.0 == x ) {
        return 1.0;
    }

    // The general case
    return std::sin( 3.1415926535897932*x )/
                    (3.1415926535897932*x);
}
```


Boolean-valued functions

- Conditions need not be conditional operators:

```
bool is_divsible( int m, int n );

// Is the argument 'm' divisible by the argument 'n'?
//   - If the remainder is zero, 'm' is divisible by 'n',
//     otherwise, it is not.
bool is_divsible( int m, int n ) {
    return ((m % n) == 0);
}
```


Boolean-valued functions

- We can then use this in a conditional statement:

```
int factor_out_bad_luck_once( int n );

int factor_out_bad_luck_once( int n ) {
    if ( is_divisible( n, 13 ) ) {
        return n/13;
    } else {
        return n;
    }
}
```


Recursive function calls

- Recall our fast sine function:
  - This returns a reasonable approximation if $0 \leq x \leq \pi/2$

```
double fast_sin( double x ) {
    return ((
        -0.11073981636184074*x - 0.057385341027109429
    )*x + 1.0)*x;
}
```

- What happens if $-\pi/2 \leq x < 0$?
  - Recall that $\sin(-x) = -\sin(x)$ or the equivalent statement $\sin(x) = -\sin(-x)$

Recursive function calls

- With the property that $\sin(x) = -\sin(-x)$, we may proceed as follows:

```
double fast_sin( double x ) {
    if ( x < 0.0 ) {
        return -fast_sin( -x );
    }

    return ((
        -0.11073981636184074*x - 0.057385341027109429
    )*x + 1.0)*x;
}
```

- This is called a recursive function call
  - If `fast_sin` is called with the argument `-0.3`, the first condition is true
    - This first condition calls `fast_sin( -(-0.3) ) == fast_sin( 3.0 )`, so now the first condition is false

Summary

- Following this lesson, you now:
  - Understand the format of a conditional statement:
    - A Boolean-valued condition,
    - a consequent block of statements to be executed if the condition is `true`, and
    - an alternative block of statements to be executed if the condition is `false`
  - We used simple, nested and cascading conditional statements to implement numerous functions

References

[1] Wikipedia
https://en.wikipedia.org/wiki/Conditional_(computer_programming)

Colophon

These slides were prepared using the Georgia typeface. Mathematical equations use Times New Roman, and source code is presented using `Consolas`.

The photographs of lilacs in bloom appearing on the title slide and accenting the top of each other slide were taken at the Royal Botanical Gardens on May 27, 2018 by Douglas Wilhelm Harder. Please see

https://www.rbg.ca/

for more information.



Disclaimer

These slides are provided for the ECE 150 *Fundamentals of Programming* course taught at the University of Waterloo. The material in it reflects the authors' best judgment in light of the information available to them at the time of preparation. Any reliance on these course slides by any party for any other purpose are the responsibility of such parties. The authors accept no responsibility for damages, if any, suffered by any party as a result of decisions made or actions based on these course slides for any other purpose than that for which it was intended.