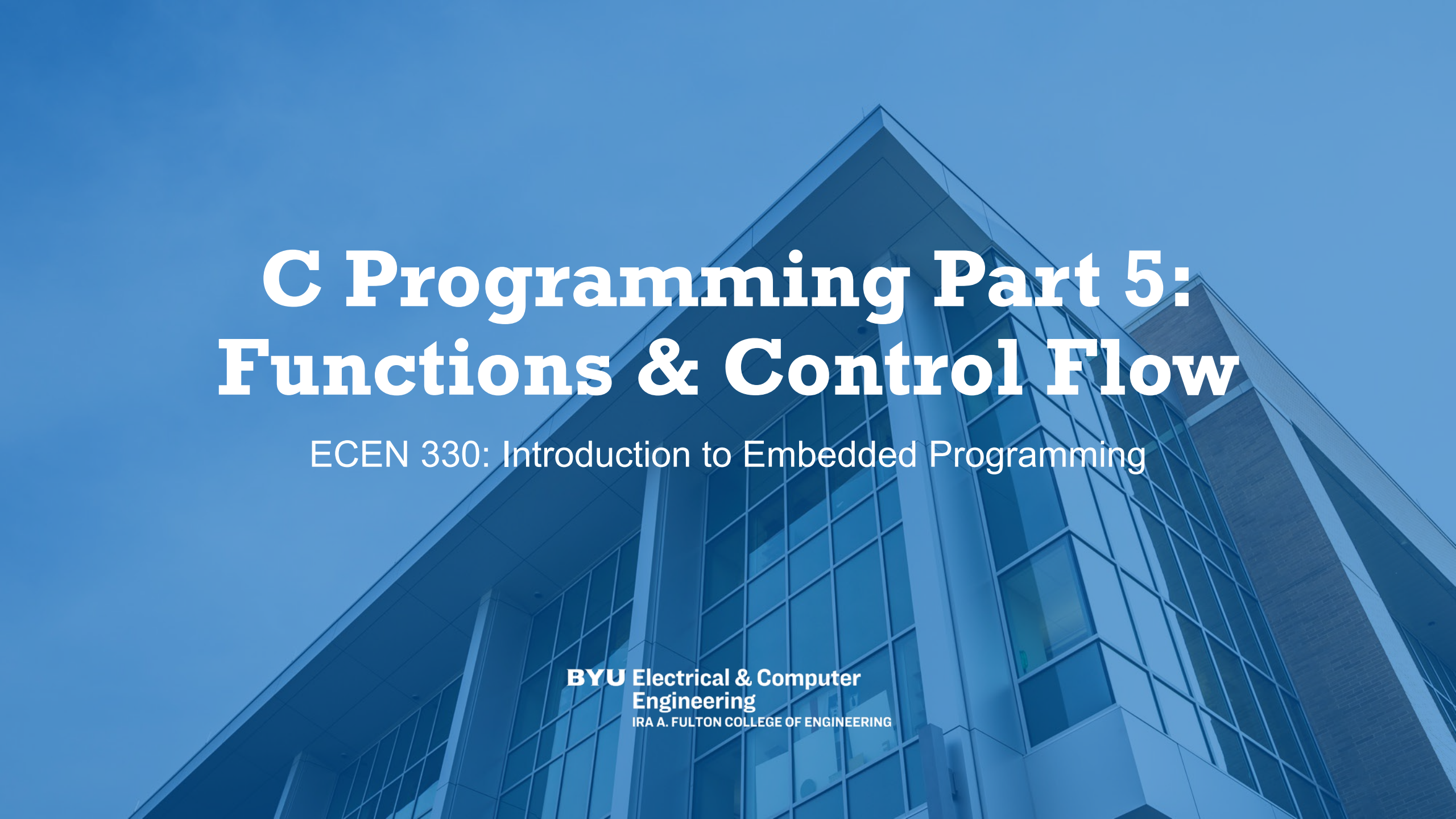

C Programming Part 5: Functions & Control Flow

ECEN 330: Introduction to Embedded Programming

BYU Electrical & Computer Engineering
IRA A. FULTON COLLEGE OF ENGINEERING

Basics of Functions

- Each *function definition* has the form

```
return-type function-name(argument declarations)
{
     declarations and statements
}
```

- Minimal function

```
dummy() {} // int return type is assumed when omitted
```

- The *return statement* is used to return a value from a function

```
return expression; // converted to the return type if needed
return; // omit expression if return type is void
```

- The calling function is free to ignore the returned value

Functions Returning Non-integers

- Must declare the return type if it is not `int`

```
/* atof: convert string s to double */
double atof(char s[])
{
        double val, power;
        int i, sign;
        ...
        return sign * val / power;
}
```

- Caller must know that a function returns non-int value

```
double atof(char []); // declaration
```

What happens if a separate file uses `atof` without a declaration?

Statements and Blocks

- An expression becomes a *statement* when it is followed by a semicolon

```
x = 0;
i++;
printf(...);
```

- Braces { and } are used to group declarations and statements together into a compound statement, or *block*

```
void foo(char s[], char t[])
{
        int i, j;
        i = strlen(s); j = 0;
        while (s[i++] = t[j++]) ;
        s[i] = '\0';
}
```

If-Else

- The `if-else` statement is used to express decisions

```
if (expression)
     statement1
else
     statement2
```

The *expression* is evaluated first

Executed if expression is non-zero

The else part is optional

Executed if expression is zero

- Ambiguity may result with a nested if sequence

```
if (n > 0)
     if (a > b)
          z = a;
else
     z = b;
```

Which `if` is the `else` associated with?

The closest previous `else`-less if

Else-If

- The `else-if` is useful for multi-way decisions

```
if (expression)
     statement
else if (expression)
     statement
else if (expression)
     statement
else if (expression)
     statement
else
     statement
```

The *expressions* are evaluated in order

If an *expression* is true, the associated statement is executed

A *statement* can be a block with braces

Switch

- The `switch` statement is a multi-way decision that tests whether an expression matches one of a number of *constant* integer values, and branches accordingly.

```
switch (expression) {
      case const-expr:
        statements
      case const-expr:
        statements
      default:
        statements
}
```

The `break` statement causes an immediate exit from the `switch`

The `default` label is executed if none of the other cases are satisfied

- What happens if a case does not end with a `break`?
- What happens if there is no default and no cases satisfied?
- Does the order of the cases matter (including `default`)?

Loops – While and For

- The `while` statement

```
while (expression)
    statement
```

Tests the termination condition at the top

- The `for` statement

```
for (expr1; expr2; expr3)
    statement
```

The advantage is centralized loop control

Any *expression* can be omitted

```
for (;;) {
    ... // infinite loop
}
```

- is equivalent to

```
expr1;
while (expr2) {
    statement
    expr3;
}
```

The comma operator (,) is useful in `for` loops

```
for (i = 0, j = strlen(s)-1; i < j; i++, j--) {
    ...
}
```

Loops – Do-While

- The syntax of the `do` is

```
do
      statement
while (expression);
```

Tests the termination condition at the bottom after each pass through the loop body

The body is always executed at least once

When the *expression* becomes false, the loop terminates

- Example

```
do {
      c = getchar();
} while (c == ' ');
```

The *statement* is executed, then *expression* is evaluated

Break

- A `break` causes the innermost enclosing loop or `switch` to be exited immediately

```
/* trim: remove trailing space */
int trim(char s[])
{
    int n;
    for (n = strlen(s)-1; n >= 0; n--)
        if (s[n] != ' ')
            break;
    s[n+1] = '\0';
    return n;
}
```

Continue

- A `continue` causes the next iteration of the enclosing loop to begin

```
for (i = 0; i < n; i++) {
    if (a[i] < 0) /* skip negative elements */
        continue;
    /* do positive elements */
    ...
}
```

Goto and Labels

- Jump immediately to the label

```
int foo( ... )
{
     for ( ... )
          for ( ... ) {
               ...
               if (disaster)
                    goto error;
          }
     ...
error:
     /* clean up the mess */
}
```

Useful to abandon processing in some deeply nested structure

A *label* is followed by a colon and can be attached to any statement in the same function as the `goto`