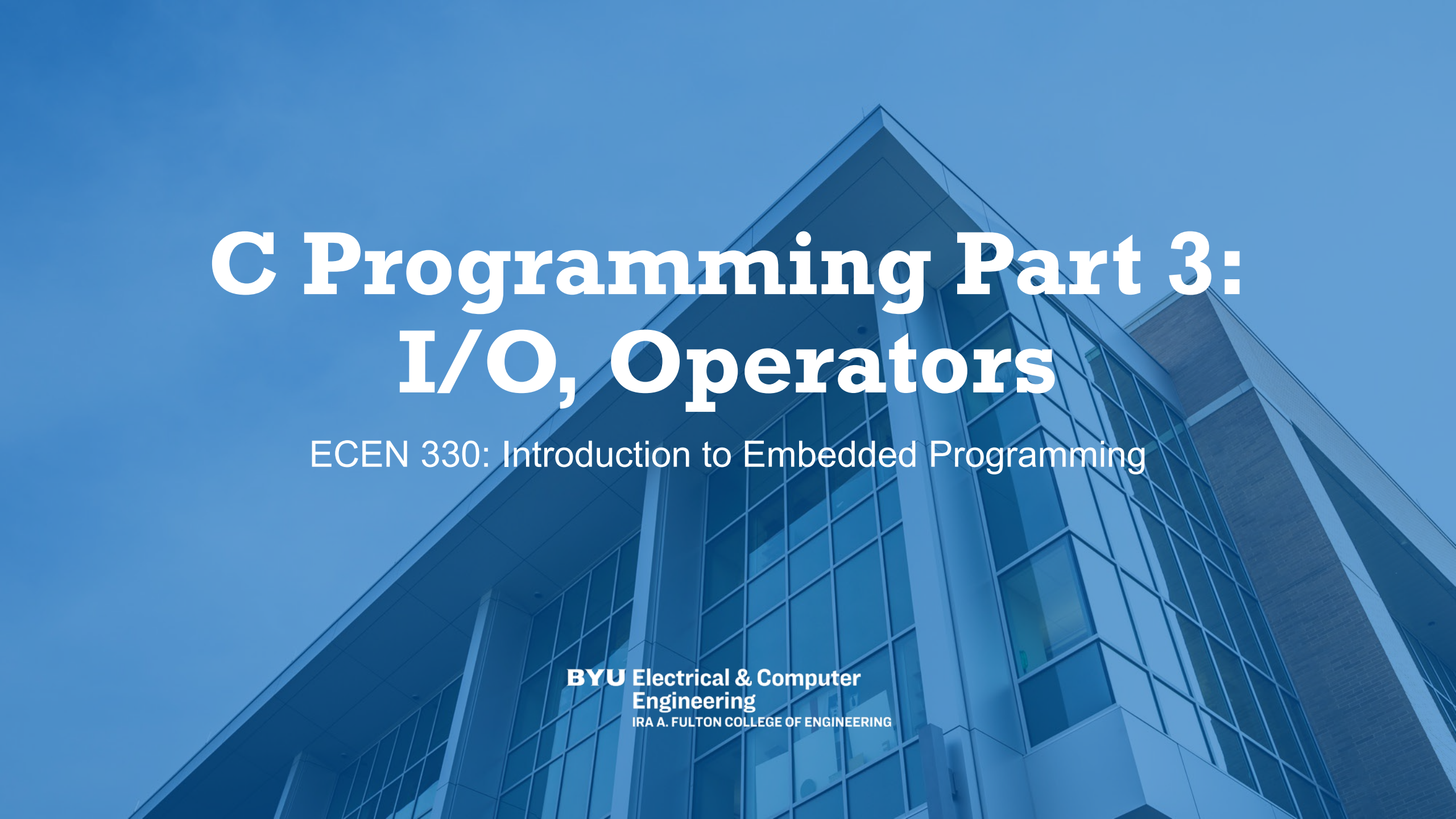

C Programming Part 3: I/O, Operators

ECEN 330: Introduction to Embedded Programming

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

Basic `printf` Conversion Specification

- Begin with % and end with a conversion character
- Between the % and conversion character, in order
  - Flags
    - -, left adjustment
    - 0, padding with leading zeros
  - Field width
    - number, minimum number of characters, pad if necessary
  - Period
    - Separates field width from the precision
  - Precision
    - number, maximum number of characters to be printed from a string, integer, or after a decimal point in a float
    - *, take value from next argument
  - Type modifier
    - h, for `short` argument
    - l, (letter ell) for `long` argument

```
printf("count:%5.3ld\n", cnt);
```

Basic `printf` Conversions

Character	Argument type; Printed As
`d,i`	`int;` decimal number
`o`	`int;` unsigned octal number (without a leading zero)
`x,X`	`int;` unsigned hexadecimal number (without a leading `0x` or `0X`), using `abcdef` or `ABCDEF` for 10, ...,15.
`u`	`int;` unsigned decimal number
`c`	`int;` single character
`s`	`char *;` print characters from the string until a `'\0'` or the number of characters given by the precision.
`f`	`double;` [-]*m.dddddd*, where the number of *d*'s is given by the precision (default 6).
`e,E`	`double;` [-]*m.dddddde+/-xx* or [-]*m.ddddddE+/-xx*, where the number of *d*'s is given by the precision (default 6).
`g,G`	`double;` use `%e` or `%E` if the exponent is less than -4 or greater than or equal to the precision; otherwise use `%f`. Trailing zeros and a trailing decimal point are not printed.
`p`	`void *;` pointer (implementation-dependent representation).

String `printf` Examples


```
printf("%s", "hello, world"); // first example
```

12 characters

Format string

```
:%s:          :hello, world:
:%10s:        :hello, world:
:%.10s:       :hello, wor:
:%-10s:       :hello, world:
:%.15s:       :hello, world:
:%-15s:       :hello, world   :
:%15.10s:     :     hello, wor:
:%-15.10s:    :hello, wor     :
```

Colons around fields to see the extent

Printf Types

- Because printf can accept any types of arguments, it **does not do any automatic casting of type for the arguments**.

- Make sure you provide the correct type for your format specifier:

```
printf("My string: %s\n", "hello there");

printf("My char: %c\n", 'y');

printf("My char: %c\n", 121);

printf("My int: %d\n", 121);
```

Basic `scanf` Conversion Specification

- Ordinary characters expected to match input stream
- Begin with % and end with a conversion character
- Between the % and conversion character, in order
  - Flags
    - *, assignment suppression
  - Field width
    - number, maximum number of characters
  - Type modifier
    - h, for `short` argument
    - l, (letter ell) for `long` or `double` argument

```
scanf("count:%5ld\n", &cnt);
```

Basic `scanf` Conversions

Character	Input Data; Argument Type
`d`	decimal number; `int *`
`i`	integer; `int *`. The integer may be in octal (leading `0`) or hex (leading `0x` or `0X`).
`o`	octal integer (with or without leading zero); `int *`
`u`	unsigned decimal integer; `unsigned int *`
`x`	hexadecimal integer (with or without leading `0x` or `0X`); `int *`
`c`	characters; `char *`. The next input characters (default 1) are placed at the indicated spot. The normal skip-over white space is suppressed; to read the next non-white space character, use `%1s`
`s`	character string (not quoted); `char *`, pointing to an array of characters long enough for the string and a terminating `'\0'` that will be added.
`e,f,g`	floating-point number with optional sign, optional decimal point and optional exponent; `float *`

Basic `scanf` Example

- Suppose we want to read

```
25 Dec 1988
```

- The scanf statement is

```
int day, year;
char monthname[20];

scanf("%d %s %d", &day, monthname, &year);
```

- No `&` is used with monthname, since an array name is a pointer.

Arithmetic Operators

- Binary arithmetic operators
  `+ -  * / %` (higher precedence)
- Unary arithmetic operators (higher precedence)
  `+ -`

- Integer division truncates any fractional part
- The `%` operator cannot be applied to a `float` or `double`

Relational and Logical Operators

- Relational operators

  `> >= < <=`

- Equality operators (lower precedence)

  `== !=`

- Logical operators (evaluated left to right)

  `&& ||`

  - Evaluation stops as soon as the truth or falsehood of the result is known

```
i < lim-1 && (c=getchar()) != '\n'
x || ++y
```

- Unary negation operator

  `!`

  `if (!valid)…` same as `if (valid == 0)…`

RH side may not be evaluated.
Important when side effects are involved.

Type Conversion

- For binary operators with operands of different types
  - "lower" type is promoted to "higher" type before operation
  - (lower) int → long → float → double (higher)
  - (lower) unsigned → signed (higher)

```
float r, f; int i;
r = i * f; /* i is converted to a float first */
```

See K&R Appendix A.6 for details

- Across assignments
  - Value on the right is converted to type on the left
  - May involve extension, rounding or truncation

```
d = i; /* double d; int i; */
```

- When arguments are passed to functions

```
r = sqrt(2); /* integer 2 is converted to double 2.0 */
```

- In an expression with a *cast*, *(type name) expression*

```
(double)ticks/TICKS_PER_SECOND /* ticks is a long */
```

Increment and Decrement Operators

- `++` adds 1 to its operand
- `--` subtracts 1 from its operand

- May be used as a *prefix* (++n) or *postfix* (n++) operator
  - `++n` increments `n` ***before*** its value is used
  - `n++` increments `n` ***after*** its value has been used

  If `n` is 5, then

```
x = n++; /* sets x to 5 */
x = ++n; /* sets x to 6 */
```

  In both cases, `n` becomes 6

- Useful when indexing arrays in a loop

```
while ((s[i++] = t[j++])) ; /* copy string t */
```

Bitwise Operators

- May only be applied to integral operands
  `char`, `short`, `int`, and `long`

- Mask off selective bits

```
/* set lowest 3 bits to 0 */
n = n & ~0x07;
```

- Turn bits on

```
/* set lowest 3 bits to 1 */
n = n | 0x07;
```

- Shift bits

```
n = n << 2; /* shift bits in n 2 positions left */
n = n >> x; /* fills with zeros if n unsigned */
```

Op	Function
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
~	Bitwise NOT
<<	Left shift
>>	Right shift

Assignment Operators and Expressions

- The operator += is called an *assignment operator*

```
i += 2; /* increment i by 2, same as i = i + 2; */
yypv[p3+p4] += 3; /* left side only evaluated once */
x *= y + 1; /* means x = x * (y + 1); */
```

- Most binary operators have a corresponding assignment operator op=, where op is:

```
+ - * / % << >> & ^ |
```

- If $expr_1$ and $expr_2$ are expressions, then

  $expr_1$ op= $expr_2$

  is equivalent to

  $expr_1$ = ($expr_1$) op ($expr_2$)

  except $expr_1$ is computed only once

Conditional Expressions

- A *conditional expression* is written with the ternary operator

  $expr_1$ `?` $expr_2$ `:` $expr_3$

  The expression $expr_1$ is evaluated first
  If it is non-zero, $expr_2$ is evaluated
  Otherwise $expr_3$ is evaluated

- Set `z` to the maximum of `a` and `b`

```
z = (a > b) ? a : b;
```

- Note that the *conditional expression* is indeed an expression and can be used where other expressions are used

Operator Precedence

Operators	Associativity
`() [] -> .`	left to right
`! ~ ++ -- + - * (type) sizeof`	right to left
`* / %`	left to right
`+ -`	left to right
`<< >>`	left to right
`< <= > >=`	left to right
`== !=`	left to right
`&`	left to right
`^`	left to right
`|`	left to right
`&&`	left to right
`||`	left to right
`?:`	right to left
`= += -= *= /= %= &= ^= |= <<= >>=`	right to left
`,`	left to right

Unary `&` `+`, `−`, and `*` have higher precedence than the binary forms.

Evaluated first

```
if (x & MASK == 0) ...
```

Evaluated first

```
x = y += z = w;
```