

# CSCI 135

## Variables, Expressions, and Statements

# Basic Language Constructs

Recall: each statement in an imperative language updates the value of some variable.

## Classes of Constructs:

- Declaration of variables: How does the variable map onto memory?
- Updating variable: How do we update the variable, and what do we update it with?
- I/O: How do we input and output data into the system?
- Control: How do we control which statement gets executed next?
- Modularity and Object Orientation: How do we organize the program to enable proper software engineering practices?
- Comments: Used to describe code; ignored by compiler, but code unmaintainable without good comments!

C/C++: on line beginning with `//` or surrounded by `/*, */`

# What Is a Variable?

- Storage class: How/where in memory is the variable stored?
- Type: What is the set of values for the variable?
- Name: What is the name of the variable (*i.e.*, identifier)?
- Scope: Where in the [syntactic] program is the variable visible?
- Value: Which value is currently stored in the variable?

Don't confuse these aspects with each other!

# Variable/Object Declaration

```
<storage class> <data type> <name1>,<name2>,... <initializer>;
```

The **name** is any legal identifier (but use meaningful ones):

- May contain letters, \_, and digits (but not start with digits)
- Case sensitive (so, x1 and X1 refer to two different variables/memory locations)
- Can not be the same as reserved words (keywords)

**Data types** affect how many bytes of memory the variable requires, and how the binary data is to be interpreted (e.g., does 10..110 mean the unsigned integer 26, the [signed twos complement] integer -14, the float 2.5, or the character 'a')?

**Storage classes** affect how/where the variable is stored; we will talk about later (leave blank for now to get defaults)

**Initializers** indicate the initial value (undefined if blank)

 **All variables must be declared** before they are used in the program, and are not visible outside the defining block

# Review - Binary Number Range

## Unsigned n-bit

Smallest number:  $0000\dots0_2 = 0_{10}$

Largest number:  $1111\dots1_2 = 2^{n-1} + 2^{n-2} + \dots + 2^0 = 2^n - 1$

# Review - Binary Number Range

## Unsigned n-bit

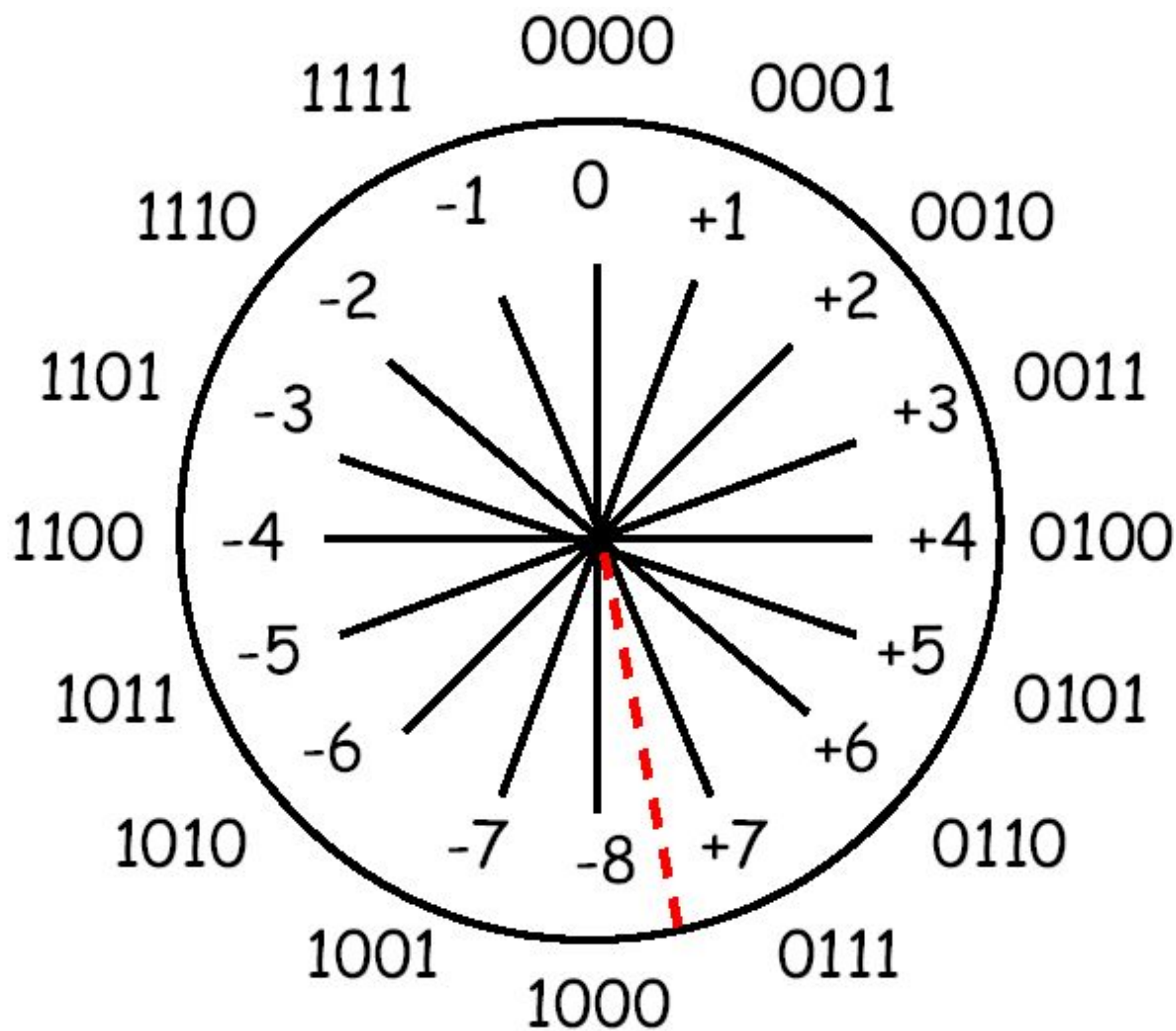
Smallest number:  $0000\dots0_2 = 0_{10}$

Largest number:  $1111\dots1_2 = 2^{n-1} + 2^{n-2} + \dots + 2^0 = 2^n - 1$

## Signed (2s complement) n-bit

Largest number:  $0111\dots1 = 2^{n-1} - 1$

Smallest number:  $1000\dots0 = -2^{n-1}$



# Some Basic Types

Name	# Bytes	Range	Interpretation
int	4	$[-2^{31}, 2^{31}-1]$	integer
unsigned int	4	$[0, 2^{32}-1]$	integer $\geq 0$
short	2	$[-2^{31}, 2^{31}-1]$	integer
unsigned short	2	$[0, 2^{32}-1]$	integer $\geq 0$
long	8	$[-2^{63}, 2^{63}-1]$	integer
unsigned long	8	$[0, 2^{64}-1]$	integer $\geq 0$
float	4	$[-10^{-38}, 10^{38}]$	real
double	8	$[-10^{-308}, 10^{308}]$	real
char	1	0-127 (NUL-DEL)	ASCII character
string (C++ only)	var.	N/A	sequence of char
bool (C++ only)	1	true, false	boolean

( $2^{31}$  is approximately  $2 \times 10^9$  or 2 billion)

Some Caveats:

- The above #bytes (thus, range) are typical but may vary by platform/compiler.
- Floats/doubles are only approximations to real numbers
- These are C++ strings; don't mix them with C strings!



# Numeric Literals

A **literal** is a specific value (of some data type).

C determines the type of the literal by its syntax:

Syntax	Literal Type	Example
Number without “.”	integer literal	2, -5, 0
Number with “.”	double literal	2., 2.5, 3.5e21
Single quotes	char literal	's'
Double quotes	string literal	“apple”

⚠ '1' is not the same as [int] 1 or “1” since they have different types and may correspond to different bit patterns in RAM

# Numeric Literals

A **literal** is a specific value (of some data type).

C determines the type of the literal by its syntax:

Syntax	Literal Type	Example
Number without “.”	integer literal	2, -5, 0
Number with “.”	double literal	2., 2.5, 3.5e21
Single quotes	char literal	's'
Double quotes	string literal	“apple”

⚠ '1' is not the same as [int] 1 or “1” since they have different types and may correspond to different bit patterns in RAM

❓ Some ASCII characters (0-31, 127) are non-printable. How do we handle such characters (also, chars with special meanings in C), and strings with them?

❗ C uses escape sequences. Partial list:

\n	new line	\'	'
\a	alert (system bell)	\"	“
		\\	\

# Updating the Value of a Variable

Assignment Statement: `<var> = <expr>;`

Terminology: `var` is called an lvalue and `expr` is called a rvalue (lval,rval for short).

Examples: `x=3*5; y=y+x;`

Semantics:

- 1 Evaluate the expression, `expr`
- 2 Store the result in the memory location corresponding to variable, `var`.

❓ What if the types of the rval and lval don't match (e.g., storing a float in an int variable)?

# Updating the Value of a Variable

Assignment Statement: `<var> = <expr>;`

Terminology: `var` is called an lvalue and `expr` is called a rvalue (lval, rval for short).

Examples: `x=3*5; y=y+x;`

Semantics:

- 1 Evaluate the expression, `expr`
- 2 Store the result in the memory location corresponding to variable, `var`.

❓ What if the types of the rval and lval don't match (e.g., storing a float in an int variable)?

❗ Strongly typed language: compile time error

❗ C: automatic type conversion, according to some predefined rules (e.g., `int n=2.99;` would assign 2 to `x`).

⚠ Don't (always) rely on implicit type conversion!

# Statements and Expressions

A **statement** is an instruction that is executed and typically updates some variable (called a **side effect**).

An **expression** is something that is evaluated, and does not update a variable (typically).

BUT, in C/C++, every statement is also an expression that evaluates to the rval.

What does `x = (y = y+2);` do if y is initially 5?

# Statements and Expressions

A **statement** is an instruction that is executed and typically updates some variable (called a **side effect**).

An **expression** is something that is evaluated, and does not update a variable (typically).

BUT, in C/C++, every statement is also an expression that evaluates to the rval.

What does `x = (y = y+2);` do if `y` is initially 5?

- 1 Evaluate `5+2` and store 7 in `y`

# Statements and Expressions

A **statement** is an instruction that is executed and typically updates some variable (called a **side effect**).

An **expression** is something that is evaluated, and does not update a variable (typically).

BUT, in C/C++, every statement is also an expression that evaluates to the rval.

What does `x = (y = y+2);` do if `y` is initially 5?

- 1 Evaluate `5+2` and store 7 in `y`
- 2 Above also resulted in evaluating the `y=y+2` expression, to 7.  
→ store 7 in `x`.

⚠ VERY BAD idea to use this (and more likely, results from a typo that will take hours to debug).

# Arithmetic Operators (and Shorthands)

- Standard arithmetic operators (e.g.,  $+$ ,  $-$ ,  $/$ ,  $*$ ,  $\%$  (mod)) (MDAS order of precedence; use parentheses for clarity)
- `lval += rval`: shorthand for `lval = lval + rval` (and similarly for `-=`, `*=`, etc.).

Ex: `x += 2`  $\equiv$  `x=x+2`)

Recommended not to use (but need to know to understand existing software).

- Post-/Pre- Increment/Decrement:
  - `x++`;: Expression evaluates to value of `x`; then adds 1 to `x`;
  - `++x`;: Expression adds 1 to `x`; then evaluates [new] value of `x`
  - Similarly for `--x`; and `x--`;

❓ If `m` is initially 5 what do `n = 2*(m++)` and `n = 2*(++m)` store in `n`?



# Arithmetic Operators (and Shorthands)

- Standard arithmetic operators (e.g.,  $+$ ,  $-$ ,  $/$ ,  $*$ ,  $\%$  (mod)) (MDAS order of precedence; use parentheses for clarity)
- `lval += rval`: shorthand for `lval = lval + rval` (and similarly for `-=`, `*=`, etc.).

Ex: `x += 2`  $\equiv$  `x=x+2`)

Recommended not to use (but need to know to understand existing software).

- Post-/Pre- Increment/Decrement:
  - `x++`;: Expression evaluates to value of `x`; then adds 1 to `x`;
  - `++x`;: Expression adds 1 to `x`; then evaluates [new] value of `x`
  - Similarly for `--x`; and `x--`;

❓ If `m` is initially 5 what do `n = 2*(m++)` and `n = 2*(++m)` store in `n`?

❗ 10 and 12, respectively (both also assign 6 to `m`)

❓ What does `n=m*(m++)` store in `n`?

# Arithmetic Operators (and Shorthands)

- Standard arithmetic operators (e.g.,  $+$ ,  $-$ ,  $/$ ,  $*$ ,  $\%$  (mod)) (MDAS order of precedence; use parentheses for clarity)
- `lval += rval`: shorthand for `lval = lval + rval` (and similarly for `-=`, `*=`, etc.).

Ex: `x += 2`  $\equiv$  `x=x+2`)

Recommended not to use (but need to know to understand existing software).

- Post-/Pre- Increment/Decrement:
  - `x++`; Expression evaluates to value of `x`; then adds 1 to `x`;
  - `++x`; Expression adds 1 to `x`; then evaluates [new] value of `x`
  - Similarly for `--x`; and `x--`;

❓ If `m` is initially 5 what do `n = 2*(m++)` and `n = 2*(++m)` store in `n`?

❗ 10 and 12, respectively (both also assign 6 to `m`)

❓ What does `n=m*(m++)` store in `n`?

❗ Either 25 or 30. C does not guarantee any order of evaluation!

⚠ Don't use `++`, `--` in larger expressions.  10/1

# Precedence vs Evaluation Order

- **Order of Precedence:** a compile-time concept that indicates how an expression is parsed (e.g., MDAS)
- **Evaluation Order:** a run-time concept that indicates order of executing expression.

Ex:  $x - (z++) * 2$

- Precedence order: the compiler generates assembly code corresponding to  $(x - (z*2))$  to evaluate the expression. It also generates code corresponding to  $z = z+1$ .
- Evaluation order:  $z$  is assigned  $z+1$  *after*  $z$  is evaluated. C/C++ does not specify whether the assignment occurs immediately after  $z$  is evaluated, after evaluating the complete expression, or somewhere in between. [▶ C++11 rules](#)


⇒ the value of  $z - (z++) * 2$  is undefined

# Each Digit in a Number Has a Weight

you multiply the weight times the digit, then add them up

decimal (base 10): each digit has a weight that is a power of 10

1000	100	10	1	weight
<b>3</b>	<b>2</b>	<b>0</b>	<b>7</b>	



$$3 \times 1000 + 2 \times 100 + 0 \times 10 + 7 \times 1$$

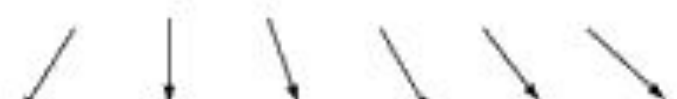
Or,  $3000 + 200 + 0 + 7 = 3207$

## Powers of 10

$$\begin{aligned} 10^0 &= 1 \\ 10^1 &= 10 \\ 10^2 &= 100 \\ 10^3 &= 1000 \end{aligned}$$

binary (base 2): each digit has a weight that is a power of 2

32	16	8	4	2	1	weight
<b>1</b>	<b>0</b>	<b>1</b>	<b>1</b>	<b>0</b>	<b>1</b>	



$$1 \times 32 + 0 \times 16 + 1 \times 8 + 1 \times 4 + 0 \times 2 + 1 \times 1$$

Or,  $32 + 0 + 8 + 4 + 0 + 1 = 45$

## Powers of 2

$$\begin{aligned} 2^0 &= 1 \\ 2^1 &= 2 \\ 2^2 &= 4 \\ 2^3 &= 8 \\ 2^4 &= 16 \\ 2^5 &= 32 \end{aligned}$$

## Convert decimal 22 to binary

largest power of 2 < or = to 22  
is 16. Put 1 at weight 16

0	0	0	1	0	0	0	0
128	64	32	16	8	4	2	1

Subtract 16:  $22 - 16 = 6$

largest power of 2 < or = to 6  
is 4. Put 1 at weight 4

0	0	0	1	0	1	0	0
128	64	32	16	8	4	2	1

Subtract 4:  $6 - 4 = 2$

The largest power of 2 < or = to 2  
is 2. Put 1 at weight 2

0	0	0	1	0	1	1	0
128	64	32	16	8	4	2	1

Subtract 2:  $2 - 2 = 0$  done

**check:**  $16 + 4 + 2 = 22$

**decimal 22 = 00010110**

### Powers of 2

$$2^0 = 1$$

$$2^1 = 2$$

$$2^2 = 4$$

$$2^3 = 8$$

$$2^4 = 16$$

$$2^5 = 32$$

$$2^6 = 64$$

$$2^7 = 128$$

$$2^8 = 256$$

$$2^9 = 512$$

$$2^{10} = 1024$$

$$2^{11} = 2048$$

$$2^{12} = 4096$$

# Convert From Binary to Decimal

128 64 32 16 8 4 2 1 weight

0 0 1 0 0 1 1 0

32 + 4 + 2 = 38 decimal

128 64 32 16 8 4 2 1 weight

1 1 0 0 1 1 0 1

128 + 64 + 8 + 4 + 1 = 205 decimal

Powers of 2

$$2^0 = 1$$

$$2^1 = 2$$

$$2^2 = 4$$

$$2^3 = 8$$

$$2^4 = 16$$

$$2^5 = 32$$

$$2^6 = 64$$

$$2^7 = 128$$

$$2^8 = 256$$

01001000 &

10111000 =

-----

00001000

01001000 |

10111000 =

-----

11111000

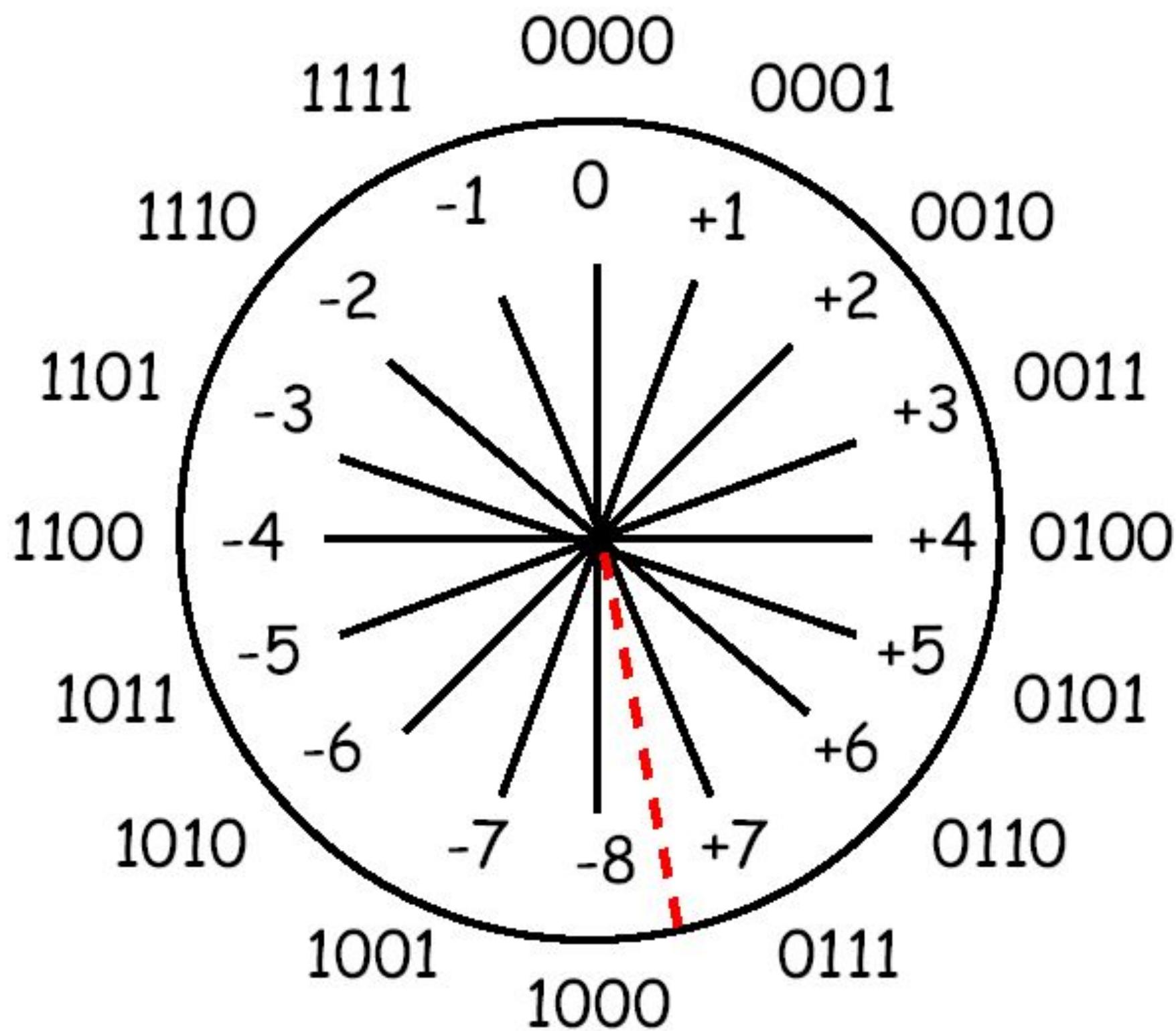
01110010 ^

10101010

-----

11011000





# Bit Operators

- $\&$ : Bit-wise and

Ex:  $1 \& 3 = 1$        $1 \& -1 = 1$

- $|$ : Bit-wise or

Ex:  $1 | 2 = 3$        $1 | -1 = -1$

- $\wedge$ : Bit-wise xor

Ex:  $1 \wedge 2 = 3$        $1 \wedge -1 = -2$

- $\sim$ : Bit-wise negation

Ex:  $\sim -1 = 0$

- $\ll$ : Shift left

Ex:  $1 \ll 2 = 4$        $1 \ll 7 = -128$  (for 8-bit ints)

- $\gg$ : Shift right

Ex:  $4 \gg 2 = 1$

All examples above assumed int (not uns)



# Bit Masks

Some properties (where  $x$  is a bit):

$$x \& 0 = 0 \quad x \& 1 = x$$

$$x \mid 0 = x \quad x \mid 1 = 1$$

$$x \wedge 0 = x \quad x \wedge 1 = \sim x$$

# Bit Masks

Some properties (where  $x$  is a bit):

$$x \& 0 = 0 \quad x \& 1 = x$$

$$x \mid 0 = x \quad x \mid 1 = 1$$

$$x \wedge 0 = x \quad x \wedge 1 = \sim x$$

Ex:

- Clear bit 0 of  $x$ :

# Bit Masks

Some properties (where  $x$  is a bit):

$$x \& 0 = 0 \quad x \& 1 = x$$

$$x \mid 0 = x \quad x \mid 1 = 1$$

$$x \wedge 0 = x \quad x \wedge 1 = \sim x$$

Ex:

- Clear bit 0 of  $x$ :  $x = x \& \sim 1$

# Bit Masks

Some properties (where  $x$  is a bit):

$$x \& 0 = 0 \quad x \& 1 = x$$

$$x \mid 0 = x \quad x \mid 1 = 1$$

$$x \wedge 0 = x \quad x \wedge 1 = \sim x$$

Ex:

- Clear bit 0 of  $x$ :  $x = x \& \sim 1$
- Set bits 2,3 of  $x$ :

# Bit Masks

Some properties (where  $x$  is a bit):

$$x \& 0 = 0 \quad x \& 1 = x$$

$$x \mid 0 = x \quad x \mid 1 = 1$$

$$x \wedge 0 = x \quad x \wedge 1 = \sim x$$

Ex:

- Clear bit 0 of  $x$ :  $x = x \& \sim 1$
- Set bits 2,3 of  $x$ :  $x = x \mid 12$

# Bit Masks

Some properties (where  $x$  is a bit):

$$x \& 0 = 0 \quad x \& 1 = x$$

$$x \mid 0 = x \quad x \mid 1 = 1$$

$$x \wedge 0 = x \quad x \wedge 1 = \sim x$$

Ex:

- Clear bit 0 of  $x$ :  $x = x \& \sim 1$
- Set bits 2,3 of  $x$ :  $x = x \mid 12$
- Flip bit 1 of  $x$ :

# Bit Masks

Some properties (where  $x$  is a bit):

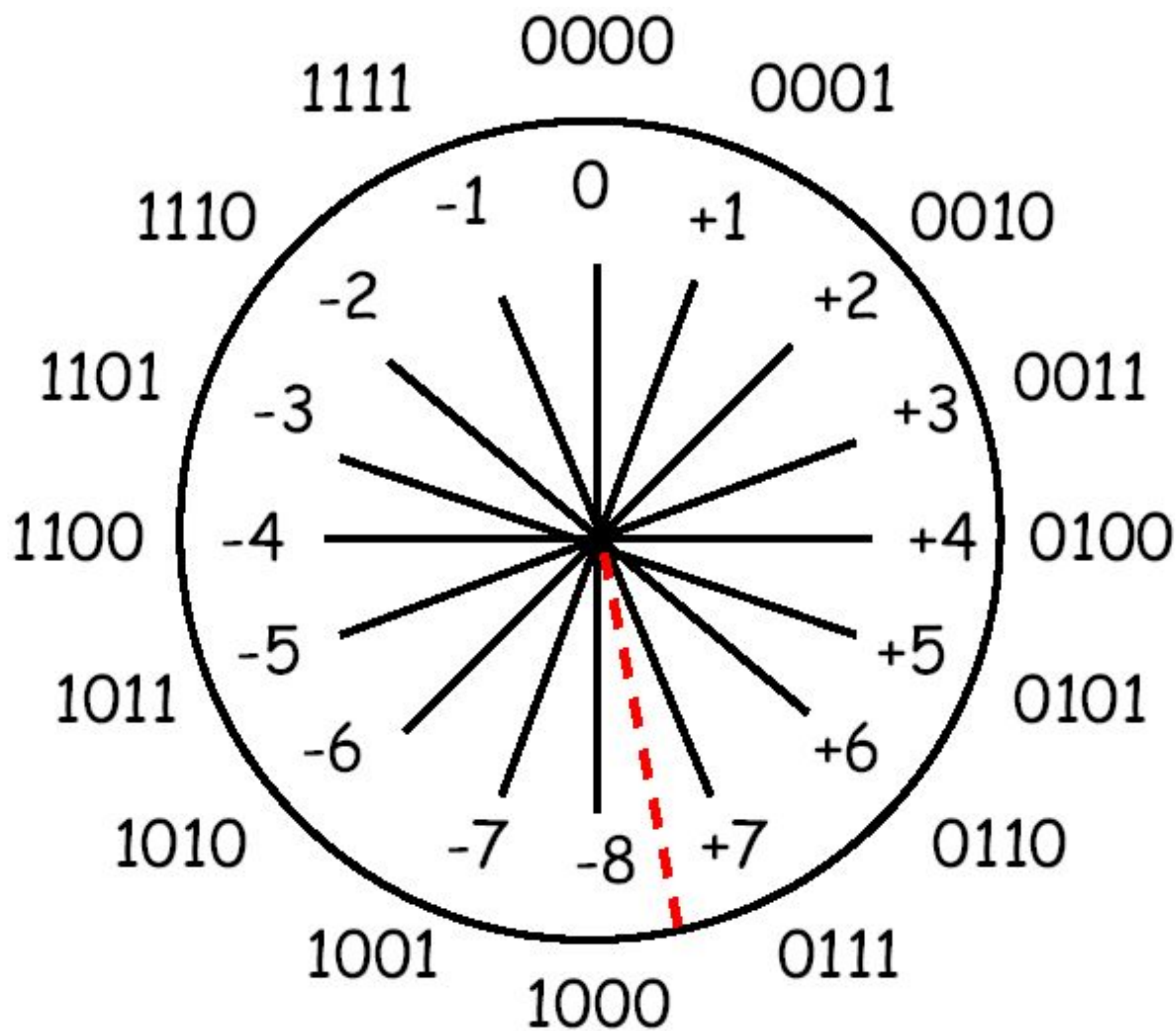
$$x \& 0 = 0 \quad x \& 1 = x$$

$$x \mid 0 = x \quad x \mid 1 = 1$$

$$x \wedge 0 = x \quad x \wedge 1 = \sim x$$

Ex:

- Clear bit 0 of  $x$ :  $x = x \& \sim 1$
- Set bits 2,3 of  $x$ :  $x = x \mid 12$
- Flip bit 1 of  $x$ :  $x = x \wedge 2$





# Some Other Operators

## String Operators (partial list)

- `+`: Concatenation  
Ex: `"apple" + "pear" = "applepear"`
- `s[i]`: *i*'th character of string *s* (*i* is an `int`  $\geq 0$ )  
Ex: `x[1] = 'e'` if `x="pear"`
- `s.empty()`: true iff *s* is an empty string (false otherwise)
- `s.length()`: length of *s*

## Relational (Comparison) Operators

`==`, `!=`, `<`, `>`, `<=`, `>=` (for numeric, char, string, ... types)

Numeric types: usual semantics

Char: ASCII ordering – see Appendix 3 (note `'A' < 'a'`)

String types: string comparison using standard lexicographic (*i.e.*, dictionary) ordering, with ASCII ordering for characters.

# Operator Semantics

The meaning of an operator depends on the types of its arguments!

- $11/2 = 5 \Rightarrow$  integer divide

- $11. / 2 = 11 / 2.0 = 5.5 \Rightarrow$  floating point divide

❓ Which kind of divide is used in  $11. / (4/2)$ ?

# Operator Semantics

The meaning of an operator depends on the types of its arguments!

- $11/2 = 5 \Rightarrow$  integer divide

- $11. / 2 = 11 / 2.0 = 5.5 \Rightarrow$  floating point divide

❓ Which kind of divide is used in  $11. / (4/2)$ ?

❗ Integer divide to compute  $4/2$ , and floating point divide for  $11./2$ !

❓ What happens if you try to assign  $11./2$  to an int n?

# Type Conversion

Recall: C/C++ does implicit/automatic type conversion if there is a type mismatch in an expression, but its not recommended to use when information is lost (e.g., `double`  $\rightarrow$  `int`).

## Type Casting (explicit conversion) in C

Precede an expression with a type name to do explicit conversion

Ex (x is a float, n is an int):

```
n = 3 + (int) x
```

```
n = (int) (x + (float) n)
```

⚠ Floats are truncated when cast to ints

❓ How do you round x and store in n?

# Type Conversion

Recall: C/C++ does implicit/automatic type conversion if there is a type mismatch in an expression, but its not recommended to use when information is lost (e.g., `double`  $\rightarrow$  `int`).

## Type Casting (explicit conversion) in C

Precede an expression with a type name to do explicit conversion

Ex (x is a float, n is an int):

```
n = 3 + (int) x
```

```
n = (int) (x + (float) n)
```

⚠ Floats are truncated when cast to ints

❓ How do you round x and store in n?

❗ `n = (int) (x + 0.5)` (not a perfect solution)

## Type Casting (explicit conversion) in C++

Similar to C, but use `static_cast` `<Type> (Expression)`

Ex: `n = 3 + static_cast <int> (x)`

Other types of casts exist (but aren't relevant until we get to OO)

# Assignment Statement Caveats

- ⚠ Watch out for type mismatches. Avoid C's implicit type conversion. When absolutely needed (typically for low-level code), use casting.
- ⚠ Don't use assignment statements as expressions.
- ⚠ Assignment is not equality: The `=` symbol in C is assignment and not the same as in math – e.g., “`x=x+1`” is nonsense in math, but in C it means that if `x` stores 4 before the statement, it stores 5 after the statement.

Unfortunate Reality: you may have to understand or maintain existing code that doesn't always do nice things.

# Exercises

- 1 Express the following decimal numbers as 8-bit unsigned binary numbers: 0, 29, 114, 218, 255
- 2 Express the following decimal numbers as 8-bit signed twos complement binary numbers: 0, -1, -2, 114, -114, 127, -128
- 3 What are the largest and smallest representable integers in 20-bit unsigned? 20-bit signed?
- 4 Write a program that inputs a character and outputs: 1) its ASCII value, and 2) the next character. You should not need to look at an ASCII chart, though you may assume that ASCII maps characters contiguously (e.g., it maps 'b' to one more than 'a').
- 5 Test what your compiler does for the undefined expression in slide 11.
- 6 Test what your compiler does for the question at the end of slide 15.