Course Coverage

1. Machine/Assembly Language: Instruction Set Architecture is instructions & datatypes of computer hardware. C/C++ related to machine features.

2. C/C++ hardware/software/system interface topics: type implementation, modularization with multiple source file programs, functions, pointers, strings, arguments, preprocessor, etc.

3. Larger, more mature projects and software designs.
Slide 3

House Architecture One

Structure

Function
Provide sheltered living spaces for people.

Slide 4

Computer Architecture One

Structure

Function
Communicate, store and process digitally represented data under digitally stored program control.
Slide 5

Variables

int X, Y, Z;
Z = X + Y*Z;
if( Z < 0 )
{
    Z = 0;
}

store values and
all look the same in C/C++

Slide 6

Present computer\textsuperscript{a} hardware stores & manipulates &
transmits all data \textbf{digitally}: which means with on/off,
0/1, \textbf{true}/\textbf{false} (usually) electrical\textsuperscript{b} signals.

A \textbf{bit} (binary digit) is a single unit of memory or
transmission that can have only 2 possible values.

\textsuperscript{a}and much other popular electronic product
\textsuperscript{b}also optical and magnetic
A representation is a coding scheme to give meaning to bit strings (sequences).

Different kinds of data are each represented by different meanings we give to sequences of bits.

Some hardware (printers, keyboards e.g.) embodies particular representations: 01000010 makes standard printers print B

The base or radix 2 (binary) number system is used by present computers to represent non-negative integers.

```c
#include <stdio.h>
unsigned char N = 0; // 8 bits will be used
for( int i=0; i<1000; i++ )
{
    printf("%d ", N ); // Print N as decimal integer.
    N = N + 1;
}

overflows or wraps around when N = 255.
0 1 ... 253 254 255 0 1 2 3 ... 255 0 ...
252 253 254 255 0 1 2 3 ... 229 230 231

Something like this absolutely must happen since 8 bits can only distinguish
2^8 = 2 * 2 * 2 * 2 * 2 * 2 * 2 = 256 different data values.
#include <stdio.h>
signed char N = 0; // 8 bits will be used
for( int i=0; i<1000; i++ )
{
    printf("%d ", N );//Print N as decimal integer.
    N = N + 1;
}

N declared **signed** this time

0 1 2 3 ... 126 127 -128 -127 -126 -125 ...
-3 -2 -1 0 1 2 ... 126 127 -128 -127 ...
-3 -2 -1 0 1 2 ... 126 127 -128 -127 ...
-3 -2 -1 0 1 2 ... 126 127 -128 -127 ...
... -29 -28 -27 -26 -25

---

The reason for the difference is subtle:

- Standard C `printf("%d", integer value );` 
  `%d` format ALWAYS uses SIGNED interpretation.

- C uses “usual argument conversions” on (8 bit) char types to (16 or 32) bit `int` since the `printf` function doesn’t have declared argument types. `printf` is **not type-safe**!

- Bits of **unsigned** `char` variables are interpreted in **unsigned** binary for conversion to `int`.

- Bits of **signed** `char` variables are interpreted in **signed** binary for conversion to `int`. 
Reference: Harbison and Steele sec. 7.4.3 Function Calls (printf has a variable number of arguments.) and 6.3.3 The Usual Unary Conversions.

Bitwise representation

- Affects computing speed
- Affects data storage space
- Causes limitations & pitfalls programmers must know about

(Earlier) Alternatives to bits for representing numbers:

- Spoken natural language.
- Written symbol schemes:
  - Tally marks (unary, base one). |||| = 4
  - Roman numerals.
  - Decimal (base ten) radix (place value) numerals use sequences of digit symbols 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0.
1998_{\text{ten}}

\begin{align*}
&1 \times 1000 + 9 \times 100 + 9 \times 10 + 8 \times 1 \\
&1 \times B^3 + 9 \times B^2 + 9 \times B^1 + 8 \times B^0
\end{align*}

Radix or place value number system (representation) example with radix or base $B = \text{ten}$.

Base or radix 2 (binary) example

\begin{align*}
64 &= 2^6 \\
32 &= 2^5 \\
16 &= 2^4 \\
1 \times 64 + 1 \times 32 + 1 \times 16 + 0 \times 8 + 0 \times 4 + 1 \times 2 + 0 \times 1 \\
64 + 32 + 16 + 2 &= 114
\end{align*}
**Slide 15**

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>1000</td>
<td>8</td>
<td>1 1</td>
<td>1001</td>
<td>9</td>
<td>10 2</td>
<td>1010</td>
<td>ten</td>
</tr>
<tr>
<td>11 3</td>
<td>1011</td>
<td>eleven</td>
<td>100 4</td>
<td>1100</td>
<td>twelve</td>
<td>101 5</td>
<td>1101</td>
<td>thirteen</td>
</tr>
<tr>
<td>110 6</td>
<td>1110</td>
<td>fourteen</td>
<td>111 7</td>
<td>1111</td>
<td>fifteen</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Slide 16**

**Base Ten Addition**

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1</td>
<td>1</td>
<td>9</td>
<td>9</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 0 2 3</td>
<td>2 0 2 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Base Two (Binary) Addition**

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 1 1</td>
<td>1</td>
<td>1 1 0 0 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 1 0 1 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 1 0 1 0 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0</td>
<td>1 0</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0</td>
<td>1 0</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0</td>
<td>1 0</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0</td>
<td>1 0</td>
<td></td>
</tr>
</tbody>
</table>
Full Adder Truth Table:

<table>
<thead>
<tr>
<th>$C$</th>
<th>$B$</th>
<th>$A$</th>
<th>sum</th>
<th>$D_1$</th>
<th>$D_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Digital circuit for an adder of two 4-bit WORDS, hardware built from 4 Full Adder subcircuits.
- The best choice of a base conversion algorithm depends on what or who is doing the conversion.

- In computers: integers already stored in binary; hardware operations work on binary representations.

- People & calculators (usually) read, write, display, think about, or do arithmetic on decimal representations.

- Algorithms are designed and written in terms of elementary operations. Available choices of operations affect what algorithms can be used.

Binary to Decimal by Human:

1110010_{two} = 64 + 32 + 16 + 0 + 0 + 2 + 0 = 114

Polynomial Evaluation Method

\[
\sum_{i=0}^{\text{word size}-1} D_i B^i
\]

\[D_i \in \{0, 1\} \quad B = 2\]
Binary to Decimal by Computer: Computer must calculate the decimal digits.

Suppose $N = 1110010_{\text{two}}$ which we know is $114_{\text{ten}}$. Program the computer to calculate:

\[
\begin{align*}
D[0] &= N \mod 10; & \text{//get 4} \\
N &= N / 10; & \text{//get 11} \\
D[1] &= N \mod 10; & \text{//get 1} \\
N &= N / 10; & \text{//get 1} \\
D[2] &= N \mod 10; & \text{//get 1} \\
N &= N / 10; & \text{//get 0, so stop.}
\end{align*}
\]

This Division Method is also useful for Humans to convert Decimal to Binary!

---

Decimal to Binary by Subtraction of Powers

1. Initialize current number to input number.

2. Repeat until current number becomes 0:
   - Guess and subtract the largest power of $2 \leq$ current number.
   - Write a 1 for the bit representing that power, and 0 for powers you didn’t subtract.
114
-64 high bit is 1

50
-32 next bit is also 1

18
-16 another bit is 1

2 next two bits, for 8 and 4, are 0
-2 The 2 bit is 1

0 Rest of the bits are 0 (only one left).
Answer=1110010

Base sixteen radix notation: hexadecimal.
Sixteen digit symbols: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
A, B, C, D, E, F
Use the 0x or 0X prefix to disambiguate decimal from
hexadecimal numerals: 0x60 is $6 \times 16 = 96_{ten}$
Programmers like hex:

- Large numbers can be written with few digits.
- Hex. numerals are very easy to convert to/from
  binary by hand.
- Hand or calculator calculations can be done in hex.
directly.
Hexadecimal/Binary interconversion

- Learn or get skill to figure fast:
  
  | 0000 0 | 0100 4 | 1000 8 | 1100 C |
  | 0001 1 | 0101 5 | 1001 9 | 1101 D |
  | 0010 2 | 0110 6 | 1010 A | 1110 E |
  | 0011 3 | 0111 7 | 1011 B | 1111 F |

- Successive adjacent groups of 4 binary digits are translated to/from Hex. digits.

- Example: DEADC4FE
  1101(D) 1110(E) 1010(A) 1101(D) 1100(C) 0100(4) 1111(F) 1110(E)

Bitwise Logical Operators

- apply to integer type objects (char, short, int, long int and their unsigned counterparts\(^a\)) and give integer results defined in terms of bits.
- Advice: Use unsigned long int for bitstrings when 32 bits are needed. (long guarantees 32 bits from ANSI C.)

  “All unsigned types use straight binary notation, regardless of whether the signed types use 2-s complement, 1-s compl. or sign magnitude ... the sign bit is treated as an ordinary bit.” (Harbison & Steele)\(^b\)

\(^a\)also bool after conversion to int

\(^b\)C/C++ on general purpose computers uses 2-s complement integers, but other sign representations are possible.
Since the MIPS Instruction Set Architecture is (1) defined in terms of how bit fields code machine language instructions, and (2) MIPS has bitwise operations, C++ bitwise operations are useful to simulate MIPS in C++.

First, single bit “Boolean” or truth value operations:

Unary “not” (complement): (also denoted by \( \neg, \overline{x}, \sim \))

<table>
<thead>
<tr>
<th>( x )</th>
<th>( \text{not}(x) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Binary “and,” “(inclusive) or,” “exclusive or”:

(Binary means “TWO OPERANDS” here)

<table>
<thead>
<tr>
<th>( x )</th>
<th>( y )</th>
<th>( x \text{ and } y )</th>
<th>( x \text{ or } y )</th>
<th>( x \text{ xor } y )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Other Symbols: \( \wedge, \& , \cdot \) \( \lor , | , + \) \( \text{eor}, \oplus \)

Other names for logic values:

1: on, true, asserted, enabled

0: off, false, deasserted, disabled
C++ Bitwise Operators apply Boolean operations to the individual bits of the binary integer representations.

unsigned char X = 0x0C; // 0000 1100 (binary)
unsigned char Y = 0x0A; // 0000 1010
(X & Y) == 0x04 // 0000 1000 AND
(X | Y) == 0x0E // 0000 1110 OR
(X ^ Y) == 0x06 // 0000 0110 EOR
(~ X) == 0xF3 // 1111 0011 COMPL

Caution: It’s a common bug to confuse Bitwise operations with Logical AND, OR, NOT:

(X && Y) == 1 if X!=0 and Y!=0, 0 otherwise
(X || Y) == 1 if X!=0 or Y!=0, 0 otherwise
(! X) == 0 if X!=0, 1 if X==0

In C/C++, consider any non-zero int or pointer as “true,” and 0 as “false.”

(In C++, non-zero ints or pointers are converted to bool true, zero converts to false, true converts to 1 and false to 0).
**Slide 31**

**Shifts**
For integral X, AMT, AMT ≥ 0

X << AMT is X shifted Left AMT bit positions. Zero bits are shifted in from the right.

Example: (0x000F8001 << 3) == 0x007C0008

For unsigned X or X ≥ 0

X >> AMT is X shifted Right AMT bit positions. Zero bits are shifted in from the left.

Example: (0x007C0008 >> 3) == 0x000F8001

For signed X, X < 0, in X >> AMT, whether 0s or 1s are shifted in is IMPLEMENTATION DEPENDENT!!

**Slide 32**

Application to a MIPS simulator project

```c
long unsigned int IR;  //to hold MIPS instr.
const int SHOP = 32 - 6;
const int MASKOP = 0x3F << SHOP;
const int OPSPECIAL = 0 << SHOP;
const int OPLW = 35 << SHOP;
switch( IR & MASKOP ) {
    case OPSPECIAL: // handle insns with opcode==0
        break;
    case OPLW:      // handle LW instruction
        break;
    // ...
    default:        // handle illegal instruction, halt.
}```
Another style:

```c
long unsigned int IR;  // to hold MIPS instr.
const int SHOP    = 32 - 6;
const int OPSPECIAL = 0;
const int OPLW    = 35;
switch( IR >> SHOP ) {
    case OPSPECIAL:    // handle insns with opcode==0
        break;
    case OPLW:         // handle LW instruction
        break;
    // ...
    default:           // handle illegal instruction, halt.
}
```

Extracting an internal field

```c
int baseregno( long unsigned int Insn )
{
    const int SHBASE    = 5 + 16; // rt+offset bits
    const int MASKBASE  = 0x1F;
    return ( Insn >> SHBASE ) & MASKBASE ;
    // alternative:
    //const int MASKBASE = 0x1F << SHBASE;
    //return ( Insn & MASKBASE ) >> SHBASE ;
}
```
Bitwise Op. Applications

1. Extract or compose bit fields when format is externally defined. (Hardware simulation, device driver software, network packet analysis/synthesis).

2. Work on small, fixed universe subsets efficiently. (HS 7.6, ios flags, Strou. (6.2.4) and p.616-7.)

3. Efficient special case arithmetic operations:

   ```c
   if( X & 3 ) { /* X is not a multiple of 4 */
     Y = X & (~0x3FF);
     /* round down to nearest 1K multiple */
   if( (X & 1) == 0 ){ /* X is even */
   }
   ```

---

How would you sort, using about 1 Megabyte of memory, a few million 7 digit telephone numbers from a disk file once every 1/2 hour, so that we can rapidly tell if a number is assigned?

A discussion is given in Jon Bentley’s “Programming Pearls” column in the December 1999 issue of Dr. Dobb’s Journal.
Sign Extension Problem: Given a length $n + 1$ bit string interpreted with a particular representation, how do you construct a longer length $m + 1$ bit string with the SAME MATHEMATICAL VALUE with the same kind of representation?

Solutions (depend on the representation):
First copy the given $n + 1$ bits into the low order $n + 1$ positions.

- Unsigned: Fill in positions $m, m - 1, \ldots, n + 2, n + 1$ with 0s.
- 1 or 2-s comp: Fill in positions $m, m - 1, \ldots, n + 2, n + 1$ with copies of $b_n$, the sign bit.
See GM p.99-100. for mathematical justification.

Complement sign extension can be programmed with bitwise AND with a single bit mask followed by OR with another mask if the AND result ≠ 0.

Question: What about sign-magnitude and biased representations?


In the C language:
Function parameters are **ALWAYS** passed by **value**.
To give a function access to a caller’s variable, the caller must give the **address of** the variable to the function.

In the C++ language:
Function prototype declaration controls whether each parameter is passed by **value** or by **reference**.
Prototype also specifies each parameter **type**.

```c
void CPFun(int num, //VALUE parameter, type int int * pnum, //VALUE parameter, type PTR to int int & rnum );//REFERENCE parameter, type int
```
REFERENCE parameters don’t exist in C.
void make9( int * pnumber ){
    printf("I'll change %d to 9\n", *pnumber);
    // Prints 38 to 9. We must dereference our pointer.
    *pnumber = 9; // value to access caller's var.
}

main(){
    int mainsNUM;
    mainsNUM = 38;
    printf("%d\n", mainsNUM); // Prints 38.
    make9( & mainsNUM );
    printf("%d\n", mainsNUM); // Prints 9.
}

The Standard C function printf SCANS its 1st argument format string and prints it char by char except for conversion specifications which begin with %.

Each conversion specification tells printf to “grab” another parameter value and print its value in converted form.
Example: %d means interpret next parameter as an integer; generate and print the sequence of ASCII characters that displays the integer’s 2-s comp. represented value in Decimal.
Example: \%s means interpret next parameter as the address of a char array; print the sequence of ASCII characters at and after that address one by one until a char with value 0 is encountered. Then stop and go on scanning the format string.

Example: \%3s is similar to \%s except stop when 3 characters have been printed or 0 is encountered, whichever happens first.

```c
printf("\%d", 5 ); //Prints 5 OK.
printf("\%3s", 5 ); //Segmentation error, crash!
// since 5 is an illegal address to dereference
printf("\%3s", ""); //Prints the null string (nothing)
```

\textbf{scanf} processes its format string like \texttt{printf} except it matches format to input characters\(^a\) except for \textbf{conversion specifications} which begin with \%.

Each conversion specification tells \texttt{scanf} to convert input chars into the specified representation, and then store the converted data in memory at the address given by the next parameter.

Example: \%d processes input characters as long as they are ASCII decimal digits. Convert one or more digits to binary (up to the next non-digit) and store the int sized data at the address given by the next parameter.

\(^a\)except one or more whitespace chars causes any number of input whitespaces to be read and discarded
int N;
scanf("%d", &N);
    // N gets the value 3 if I input 3 <newline>.
scanf("%d", N);
    // Segmentation error, crash.
    // The current value on N, 3, is not a legal
    // memory address for scanf to dereference.
cin >> N;
    // N gets the value 3 if I input 3 <newline>.
cin >> * reinterpret_cast< * int>( N);
    // How to FORCE C++ to crash like the 2nd scanf.
    // You can do it, but it's tough!

** Slide 45 **

** char C type HS 5.1.3; (opt: Strou 20.2) **

- Holds a character, typically ASCII coded, almost
  universally 8 bits; unit for most HW memory
  addresses. Unit of size reported by C's \texttt{sizeof()} 
  operator (so chars are smallest).
- Literal values (character constants): 'a' '0'(=48)
  '\n' '\0'=0
- Built-in and converts to/from other integral types.
- Signed or unsigned varies; use "signed char" or
  "unsigned char" if your program does arithmetic
  on chars.

** Slide 46 **
Character handling library:

- **Documentation**: HS Ch. 12 *man ctype* in Lab.
- Header file: `<ctype.h>`
- Tests and manipulations on character data.
- Functions operate in *integer* data and perform arithmetic operations. C/C++ converts *char* values to/from *int* values.

Examples:

```c
int isdigit (int x);  // '0'..'9'
```

- Returns a non-zero int if the value stored in `x` represents an ASCII digit, and 0 otherwise.
int tolower (int x)

Returns the lower case equivalent of the char stored in x, if it is an upper case letter; otherwise, returns x itself.

Implementation in C++:

```c++
int tolower (int x) {
    if ( (x >= 'A')
        && (x <= 'Z')
    )
        return (x - 'A' + 'a');
    else
        return x;
}
```

Implementation in MAL:

tolower:

```assembly
move $v0, $a0
blt $v0, 65, done
bgt $v0, 90, done
addiu $v0, $v0, 32
done:         jr $ra
```

These implementations rely on the Locale Dependent! fact that the ASCII codes for the alphabet are contiguous. This is false for the EBCDIC (old IBM) code.
Table Lookup Implementation in C++:

```c
unsigned int tolower_TABLE[ 256 ] =
{ 0, 1, ..., ..., 'a', 'b', 'c', ...
  ..., ..., 'a', 'b', 'c', ...
  ..., ..., ..., 253, 254, 255 };

int tolower (int x)
{
  x = x & 0xFF; // Mask out high bits
  // above lower 8.
  return tolower_TABLE[ x ];
}
```

---

Saving Space (Strou. C.8)

1. Bit fields (HS 5.6.5): Put more than one small object into a byte.

2. Unions (HS 5.7): Use the same space to hold different (types) of objects at different times.

Bitwise (mask and shift) operations are **portable**: They denote the same mathematical operations on equal sized integers, regardless of Endianness. But other C/C++ features can help save space.
1. Bit Fields (HS 5.6.5): Specify integral fields in a `struct` with given bit lengths. The compiler tries to pack them tightly, in a non-portable way: Depends on Byte-Ordering and Alignment Boundaries.

2. Unions: Specify two or more different fields with different types or sizes that will occupy the same memory space in a `struct`. Program logic or other fields must keep track of the kind of value currently in the union.

Unions “shouldn’t” be used for type conversion, since C++ has cast operations that check the validity of the conversion.

Unions can be used to explore what data representations a system actually uses.

If you use try to access bytes by byte offsets in a `char` array overlaid on the word array, the byte selection will depend on the C++ machine’s byte order.