Learning Objectives

• Learn about floating point number
• Learn about typedef, enum, and union
• Learn and understand pointers
FLOATING POINT NUMBER
IEEE Floating Point Standard
IEEE Floating Point Standard

- sign bit
- exponent
- fraction
IEEE Floating Point Standard

- sign bit
- exponent
- fraction

\((-1)^s \times 2^E \times F\)

1. Sign bit \(s\) determines whether number is negative or positive
2. Exponent \(E\) weights value by power of two
3. Fractional part \(F\) normally a fractional value in range \([1.0,2.0)\).
IEEE Floating Point Standard

- **sign bit**
- **exponent**
- **fraction**

\[ (-1)^s \times 2^E \times F \]

1. Sign bit \( s \) determines whether number is negative or positive
2. Exponent \( E \) weights value by power of two
3. Fractional part \( F \) normally a fractional value in range \([1.0,2.0)\).
   - Since \( F \) always starts with 1, we don’t store it
IEEE Floating Point Standard

float

```
<table>
<thead>
<tr>
<th>s</th>
<th>exp</th>
<th>frac</th>
</tr>
</thead>
</table>
```

1  k=8  n=23
IEEE Floating Point Standard

float

\begin{align*}
\text{s} & \quad \text{exp} & \quad \text{frac} \\
1 & \quad k=8 & \quad n=23
\end{align*}

double

\begin{align*}
\text{s} & \quad \text{exp} & \quad \text{frac} \\
1 & \quad k=11 & \quad n=52
\end{align*}
IEEE Floating Point Standard

\((-1)^s \times 2^e \times f\)
IEEE Floating Point Standard

• when exp = 00..0, represent 0
IEEE Floating Point Standard

\[ (-1)^s \times 2^E \times F \]

- when \( \text{exp} = 00..0 \), represent 0
- when \( \text{exp} = 11..1 \)
  - \( \text{frac}=00..0 \), represent \( \infty \)
IEEE Floating Point Standard

\[ (-1)^s \times 2^e \times f \]

- when \( exp = 00..0 \), represent 0
- when \( exp = 11..1 \)
  - \( frac=00..0 \), represent \( \infty \)
  - \( frac!\neq00..0 \), represent NaN or “Not a Number”
IEEE Floating Point Standard

- when \( \text{exp} = 00..0 \), represent 0
- when \( \text{exp} = 11..1 \)
  - \( \text{frac}=00..0 \), represent \( \infty \)
  - \( \text{frac}! = 00..0 \), represent NaN or “Not a Number”
- Otherwise, \( E = \text{exp} - \text{bias} \)
  - \( \text{bias}=2^{k-1}-1 \)
  - float: \( k=8 \), bias=127

\((-1)^{s} \times 2^{E} \times F\)
Encode 72.0f in IEEE floating point standard

\[ (-1)^s \times 2^E \times F \]

Step 1: Write 72.0 in binary scientific notation

\[ 72.0 = 1001000 = 1.001000 \times 2^6 \]
Encode 72.0f in IEEE floating point standard

\[ (-1)^s \times 2^{\text{exp}} \times \text{frac} \]

\[
\begin{array}{c|c|c}
s & \text{exp} & \text{frac} \\
1 & k=8 & n=23 \\
\end{array}
\]

Step 1: Write 72.0 in binary scientific notation
72.0 = 100 1000 = 1.001000 \times 2^6

Step 2: compute fractional value:
frac = 0010 0000 0000 0000 0000 0000 0000 000
Encode 72.0f in IEEE floating point standard

<table>
<thead>
<tr>
<th>s</th>
<th>exp</th>
<th>frac</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>k=8</td>
<td>n=23</td>
</tr>
</tbody>
</table>

\((-1)^s \times 2^E \times F\)

Step 1: Write 72.0 in binary scientific notation
72.0_{10} = 100 1000_2 = 1.001000\times2^6

Step 2: compute fractional value:
frac = 0010 0000 0000 0000 0000 0000 000

Step 3: compute the exponent value:
E = exp – bias, exp = E+bias = 6 + 127 = 133_{10} = 1000 0101_2
Encode 72.0f in IEEE floating point standard

Step 1: Write 72.0 in binary scientific notation
72.0 = 100 1000 = 1.001000×2^6

Step 2: compute fractional value:
frac = 0010 0000 0000 0000 0000 0000 0000

Step 3: compute the exponent value:
E = exp − bias, exp = E+bias = 6 + 127 = 133 = 1000 0101

(-1)^s × 2^E × F

<table>
<thead>
<tr>
<th>s</th>
<th>exp</th>
<th>frac</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>k=8</td>
<td>n=23</td>
</tr>
</tbody>
</table>
TYPEDEF
Typedef

- Using structures is cumbersome
  - Everything related has to include the “struct” keyword
Typedef

- Using structures is cumbersome
  - Everything related has to include the “struct” keyword

- C allows you to give a type an alias
  - `typedef` is a keyword in C
  - Give a meaningful name to an existing type
typedef Example

• Syntax

typedef existing-type new-name

typedef unsigned char byte;

Typedef Example
typedef struct StudentRecord {
    char name[25];
    int id;
    char gender;
    double gpa;
};

typedef struct StudentRecord StudentRecord;
typedef Example

```c
struct StudentRecord {
    char name[25];
    int id;
    char gender;
    double gpa;
};

typedef struct StudentRecord StudentRecord;

// Example usage:
struct StudentRecord s;
```
typedef struct  StudentRecord
{
    char name[25];
    int id;
    char gender;
    double gpa;
} StudentRecord;
typedef struct StudentRecord
{
    char name[25];
    int id;
    char gender;
    double gpa;
} StudentRecord;
typedef struct StudentRecord
{
    char name[25];
    int id;
    char gender;
    double gpa;
} StudentRecord;

struct StudentRecord record;
typedef struct StudentRecord {
    char name[25];
    int id;
    char gender;
    double gpa;
} StudentRecord;

struct StudentRecord record;
typedef struct StudentRecord
{
    char name[25];
    int id;
    char gender;
    double gpa;
} StudentRecord;
StudentRecord record;
ENUMERATION & UNION
Enumerations in C

• **What are enumerations?**
  – A convenient construct for associating names with constant values that have a type.

• **Syntax:**

```c
enum Color { RED, GREEN, BLUE };  
enum Color color = RED;  
```
C Unions

• What is a union?
  – Like structures, but every field occupies the same region in memory!
  – The largest type in the union defines the total size of that union.
  – Use one member at a time
C Unions

• What is a union?
  – Like structures, but every field occupies the same region in memory!
  – The largest type in the union defines the total size of that union.
  – Use one member at a time

• Example:

```c
union value {
    float f;
    int i;
    char s;
};
```

```c
union value v;
v.f = 45.7;
v.i = 12;
v.s = ‘X’;
```
C Unions

Example:
#include <stdio.h>

union ufloat {
    float f;
    unsigned u;
};

int main()
{
    union ufloat ul;
    ul.f = 72.0f;
    printf("%X\n", ul.u);
}
iClicker question

union value { float f; int i; char s; };
struct value { float f; int I; char s; };
The sizes of the union and the struct are (on x64 with gcc):
A. union: 12 bytes, struct 12 bytes
B. union: 9 bytes, struct 12 bytes
C. union: 4 bytes, struct 9 bytes
D. union: 5 bytes, struct 12 bytes
E. union 4 bytes, struct 12 bytes
union.c example

• Let us compile this example
  – Compilers usually maintain information about variables, this example is the start of a data structure for doing this...
  – Note how the different types interpret the bits differently!
  – This example shows how character arrays and integers are interpreted differently!
C POINTER
C Pointers

What is a pointer?
C Pointers

What is a pointer?

A pointer is like a mailing address, it tells you where something is located.
C Pointers

What is a pointer?

A pointer is like a mailing address, it tells you where something is located.

Every object (including simple data types) in Java and C reside in the memory of the machine.
C Pointers

What is a pointer?

A pointer is like a mailing address, it tells you where something is located.

Every object (including simple data types) in Java and C reside in the memory of the machine.

A pointer to an object is an “address” telling you where the object is located in memory.
C Pointers
C Pointers

So why do I care about pointers?
C Pointers

So why do I care about pointers?

In Java, you do not have access to these pointers (or addresses).

In Java, you do not have access to the address of an object.

This provides safety!
C Pointers

So why do I care about pointers?

In Java, you do not have access to these pointers (or addresses).

In C, you do have *access to the address* of an object, which allows you to *interact with hardware*. 
A pointer is denoted by ‘*’ and has a type.

The type is the kind of thing assumed to be at the location the pointer refers to.
#include <stdio.h>

int main() {
    int *ptr;
    int *ptr2;
    int x = 2;
    int y = 5;
}

C Pointers

#include <stdio.h>

int main() {
    int *ptr;
    int *ptr2;
    int x = 2;
    int y = 5;
    ptr = &x;
    ptr2 = &y;
}

You can assign an “address” to a pointer using the “address of” (&) operator.
int x = 2;

int ptr = &x;
int x = 2;

A pointer can point to itself.

int ptr = &x;

A Visual...

A Visual...

x

2

ptr

&x
So, if `ptr` is a pointer that refers to a value in memory... How do we get the value?
C Pointers

```c
#include <stdio.h>

int main() {
    int *ptr;
    int *ptr2;
    int x = 2;
    int y = 5;
    ptr = &x;
    ptr2 = &y;
    printf("Value : *ptr = %d\n", *ptr);
    printf("Address: ptr = %d\n", ptr);
}
```

You **dereference** (follow) the pointer! (Note the * !)
C Pointers

Imagine we have the following declarations...

```c
int x;
int *ptr = &x;
```
Imagine we have the following declarations...

```c
int x;
int *ptr = &x;
```

- `x` is located “somewhere” in memory
- `ptr` is also located “somewhere” in memory
C Pointers

Imagine we have the following declarations...

```c
int x;
int *ptr = &x;
```

- `x` is located “somewhere” in memory
- `ptr` is also located “somewhere” in memory

`ptr` “points” to the location representing `x`. 
Imagine we have the following declarations...

```c
int x;
int *ptr = &x;
```

- `x` is located “somewhere” in memory
- `ptr` is also located “somewhere” in memory

`ptr` “points” to the location representing `x`.
C Pointers

Each box is a byte
C Pointers

<table>
<thead>
<tr>
<th>Memory</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
<th>21</th>
</tr>
</thead>
</table>

Each box is a byte **and** has a location.

Memory is very much like a giant character array!
C Pointers

int *ptr

int x;
int *ptr = &x;
C Pointers

```c
int *ptr

*ptr = 0;
```
C Pointers

int *ptr

*ptr = 0;
C Pointers

```c
int *ptr

*ptr = 0;
x = 10;
```
int *ptr

*ptr = 0;
x = 10;

C Pointers
C Pointers

```c
int *ptr

*ptr = 0;
x = 10;
```
C Pointers

What does this do?

```c
int *ptr = &y;
x = 10;
*ptr = 0;
```

```c
What does this do?
```
What does this do? The pointer (ptr) is assigned a different address.

```c
int *ptr

*ptr = 0;
x = 10;
ptr = &y;

What does this do?
The pointer (ptr) is assigned a different address.
```
C Pointers

```c
int *ptr

*ptr = 0;
x = 10;
ptr = &y;

*ptr = 4;
```

![Diagram](image-url)
int *ptr

*ptr = 4;

*ptr = 0;
x = 10;
ptr = &y;
Why is `ptr` floating off to the side? I thought it was *also in memory*?
Let us look at this a little more carefully...

```c
int x;
int y;
int *ptr;
x = 1;
y = 2;
ptr = &x;
*ptr = 99;
ptr = &y;
*ptr = 88;
```
Let us look at this a little more carefully...

```c
int x;
int y;
int *ptr;
x = 1;
y = 2;
ptr = &x;
*ptr = 99;
ptr = &y;
*ptr = 88;
```
Let us look at this a little more carefully...

```c
int x;
int y;
int *ptr;
x = 1;
y = 2;
ptr = &x;
*ptr = 99;
ptr = &y;
*ptr = 88;
```

C Pointers
Let us look at this a little more carefully...

```c
int x;
int y;
int *ptr;
x = 1;
y = 2;
ptr = &x;
*ptr = 99;
ptr = &y;
*ptr = 88;
```
Let us look at this a little more carefully...

```c
int x;
int y;
int *ptr;
x = 1;
y = 2;
ptr = &x;
*ptr = 99;
ptr = &y;
*ptr = 88;
```
Let us look at this a little more carefully...

```c
int x;
int y;
int *ptr;
x = 1;
y = 2;
ptr = &x;
*ptr = 99;
ptr = &y;
*ptr = 88;
```
Let us look at this a little more carefully...

```c
int x;
int y;
int *ptr;
x = 1;
y = 2;
ptr = &x;
*ptr = 99;
ptr = &y;
*ptr = 88;
```
Let us look at this a little more carefully...

```c
int x;
int y;
int *ptr;
x = 1;
y = 2;
ptr = &x;
*ptr = 99;
ptr = &y;
*ptr = 88;
```
Let us look at this a little more carefully...

```c
int x;
int y;
int *ptr;
x = 1;
y = 2;
ptr = &x;
*ptr = 99;
ptr = &y;
*ptr = 88;
```

Is this legal?

```c
*ptr = &y
```
Let us look at this a little more carefully...

What if we do this?

```c
int **dptr = &ptr;
```
Let us look at this a little more carefully...

What if we do this?

```c
int **dptr = &ptr;
```
Let us look at this a little more carefully...

What if we do this?

```c
int **dptr = &ptr;
```

```
int *ptr
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
88
16 17 18 19
99
20
```

```c
int x 20
int y 16
```
Pointer Arithmetic
Let us look at this a little more carefully...

What does this mean?

```c
int x
int y
int *ptr

ptr = ptr + 1;
```
Let us look at this a little more carefully...

What does this mean?

```c
ptr = ptr + 1;
```

1 is scaled.
Let us look at this a little more carefully...

What does this mean?

```c
ptr = ptr + 1;
```

**Pointer Arithmetic**
Let us look at this a little more carefully...

What does this mean?

\[ \text{ptr} = \text{ptr} + 1; \]

You can use \text{ptr++} or \text{++ptr}, add/subtract a constant
What is the value of $y$ after executing the following program?

```c
int x, y;
int *p;
p = &x;
x = 5;
*p = 23;
y = x+1;
```

A. 5  
B. 23  
C. 6  
D. 24
What is the value of y after executing the following program?

```c
{ int x, y, a; int *p, *q; p = &a; q = &x; x = 5; *p = 23; y = x+1; }
```

A. 5  
B. 23  
C. 6  
D. 24
Pointer operations

• Referencing: Create a location
  – type *p = &var;
Pointer operations

• Referencing: Create a location
  – type *p = &var;

• Dereferencing: Access a location
  – int x = *ptr; (read)
struct point  {
    int x;
    int y;
};

struct point  {
    int x;
    int y;
};

struct point origin, *p;
p = &origin;
printf("origin is (%d,%d)\n", (*pp).x, pp->x);
Pointer operations

- **Referencing**: Create a location
  - type *p = &var;

- **Dereferencing**: Access a location
  - int x = *ptr; (read)
  - x = ptr2->field; (read)
Pointer operations

- **Referencing**: Create a location
  - type *p = &var;

- **Dereferencing**: Access a location
  - int x = *ptr; (read)
  - x = ptr2->field; (read)
  - *ptr = x; (write)
Pointer operations

• **Referencing: Create a location**
  – type *p = &var;

• **Dereferencing: Access a location**
  – int x = *ptr; (read)
  – x = ptr2->field; (read)
  – *ptr = x; (write)
  – ptr2->field = x; (write)
Pointer operations

• **Referencing: Create a location**
  – type *p = &var;

• **Dereferencing: Access a location**
  – int x = *ptr; (read)
  – x = ptr2->field; (read)
  – *ptr = x; (write)
  – ptr2->field = x; (write)

• **Aliasing: Copy a pointer**
  – type *pa;
    pa = pb;