Types

Professor: Suman Saha

Basic Types

- Types that are not defined on top of other types
  - Examples: int, float, bool, char
  - Composite types: struct types; arrays; pointer types; function types …

Memory units for storing types

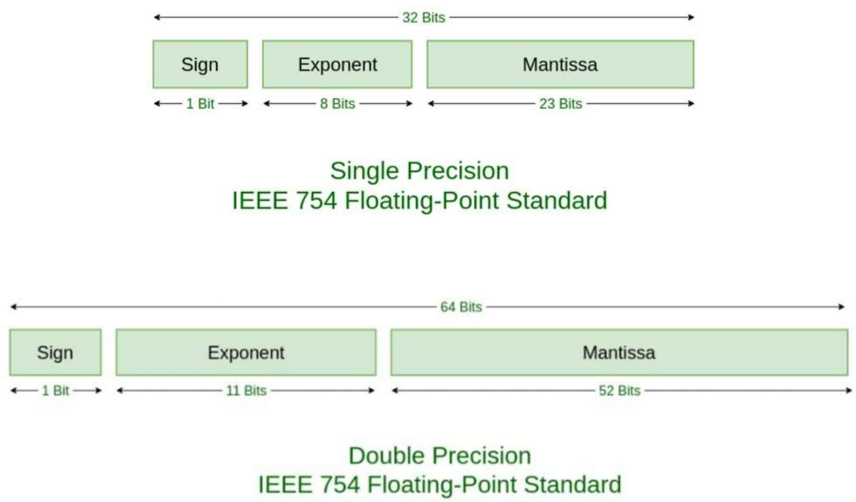
- Terminology in use with current 32-bit computers:
  - Nibble: 4 bits (a hex digit)
  - Byte: 8 bits
  - Half-word: 16 bits
  - Word: 32 bits
  - Double word: 64 bits
  - Quad word: 128 bits

Integer types

- In most languages, integer types are finite in size.
- For example, Java
  - byte: 8-bits; short: half-word; int: a word; long: a double word
- So, a + b may overflow the finite range.
  - E.g., (2^32 -1) + 1 -> 0
- There's also a difference between signed and unsigned representations

Floating point types

- Model real numbers, but only as approximations
- Single precision (float): 32 bits, double precision (double): 64 bits
- IEEE Standard 754 Floating Point Numbers are represented using three components
  - The sign of Mantissa
  - Exponent
  - Mantissa



Boolean and Character (omit)

- Boolean: true, false
  - Could be implemented as bits, but often as bytes
  - Advantage: increase readability
- Character
  - Stored as numeric codings (e.g., ASCII, ISO 8859-1, Unicode: Java, JavaScript, C#)
  - UTF-8, UTF-16, UTF-32

Enumeration types

- Enumeration (C/C++):
  - enum day {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday};
  - enum day myDay = Wednesday;
  - In C/C++ the above values of this type are 0, ..., 6.
- More powerful in Java:

```
enum day {Monday, …, Sunday};
for (day d : day.values())
   Sytem.out.println(d);
```

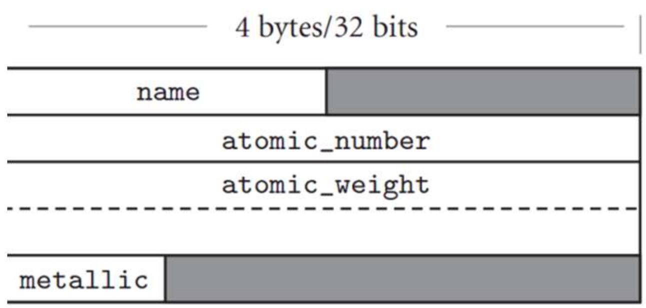
Records and Structure

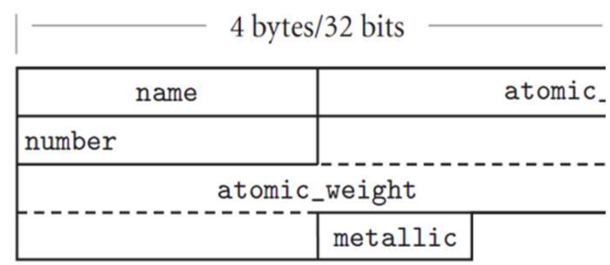
- Usually laid out contiguously
- Possible holes for alignment
- Compilers may re-arrange field to minimize holes

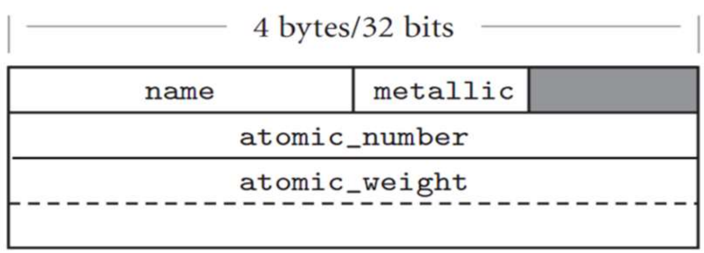
```
struct id {
  int i;
  double d;};
struct id x;
x.i, x.d
```

Records and Structure

```
struct element {
   char name[2];
   int atomic_number;
   double atomic_weight;
   _Bool metallic;}
```







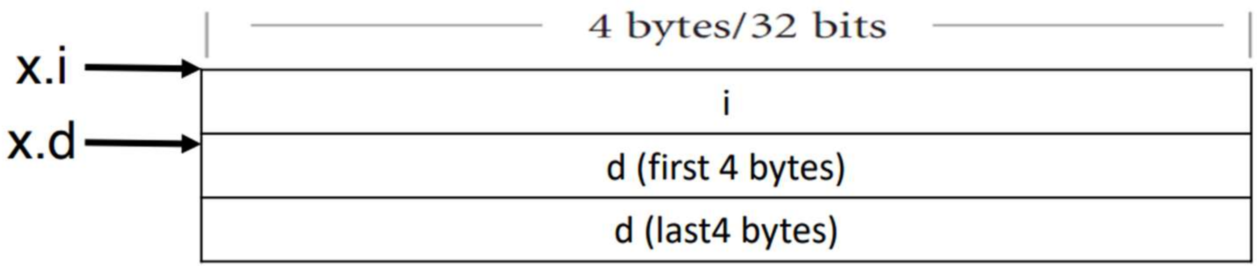
Possible
Memory Layouts

Records and Structure

Usually laid out contiguously

```
struct id {
  int i;
  double d;};
struct id x;
x.i, x.d
```

A possible memory layout:



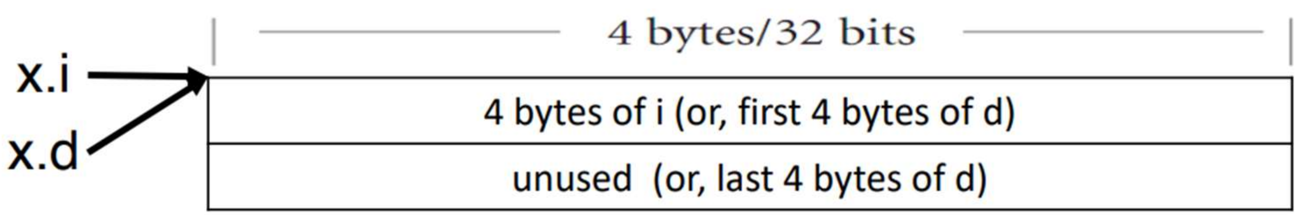
Each field has a separate piece of memory

Union Types

Laid out in shared memory

```
union id {
  int i;
  double d;};
union id x;
x.i = 1;
y = 1.0 + x.d;
```

A possible memory layout:



All fields share the same piece of memory

Union Types

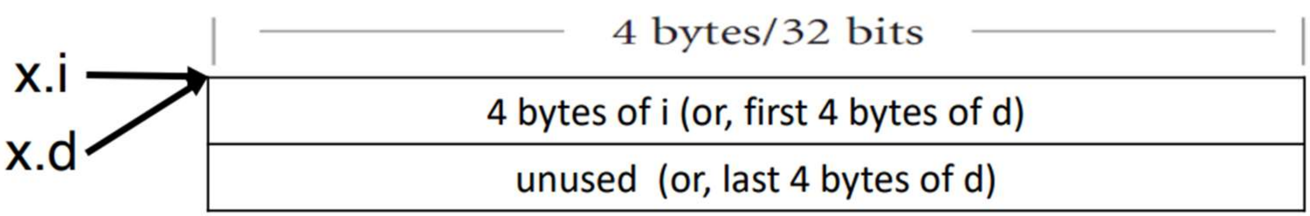
Not type safe:

x.d will read the binary

0x00000001,???????? as a double

```
union id {
  int i;
  double d;};
union id x;
x.i = 1;
y = 1.0 + x.d;
```

A possible memory layout:



How can we make it type safe?

Discriminated Union Types

A combination of a tag (like an enum) and a payload per possibility (like a union).

```
enum Kind {isInt, isFloat}
struct intorreal {
  enum Kind which;
  union U {int a; float p} u;
} ir;
float x = 1.0;
if (ir.which == isInt) ir.u.a = 1;
if (ir.which == isFloat) x = x + ir.u.p;
```

Still not type safe: type system doesn't enforce tag check.

Sum Types

Many functional programming languages support type-safe sum types

(possibly empty) payload type

Tag

Haskell

```
data intorreal = isInt Int | isFloat Float
-- given u has type intorreal
case u of
  isInt i -> i + 1
  isFloat f -> f + 1.0
```

Type safe: type is checked under each case statement (the only way to read from a value with the sum type)

Sum Types are General

Haskell

Tag

```
data day = Monday | Tuesday | Wednesday |
           Thursday | Friday | Saturday | Sunday
-- Given d has type day
case d of
  Monday -> …
  Tuesday -> …
  …
```

A generalization of Enumeration type

Sum Types and Product Types

Sum Types: alternation of types

Product Types: concatenation of types (such as?)

(records and structures are product types)

```
enum Color {Red, Blue}
enum Shape {Circle, Rectangle}
struct ColoredShape {enum color c;enum shape s}
```

Values of color

Values of shape

Values of coloredShape

Analogy:

(Red + Blue) * (Circle + Rectangle)

= (Red*Circle) + (Red*Rectangle) + (Blue*Circle) + (Blue*Rectangle)

Array

Lifetime and array size

- Global lifetime, static shape

```
int A[10];
```

- Local lifetime, static shape

```
int f() {
  int A[10];
  … }
```

- Local lifetime, Dynamic shape

```
int f(int n) {
  int A[n];
  … }
```

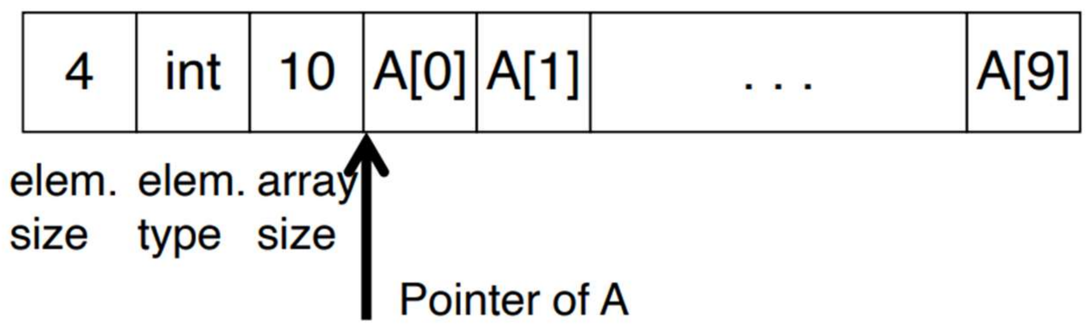
C Array

- Static/Stack/Heap allocated
- Size statically/dynamically determined
- Array bounds not checked (buffer overflow)

Java Array

- Heap allocated
- Size dynamically determined
- Array size is part of stored data (Dope Vector)
- Array bounds checked

Dope Vectors

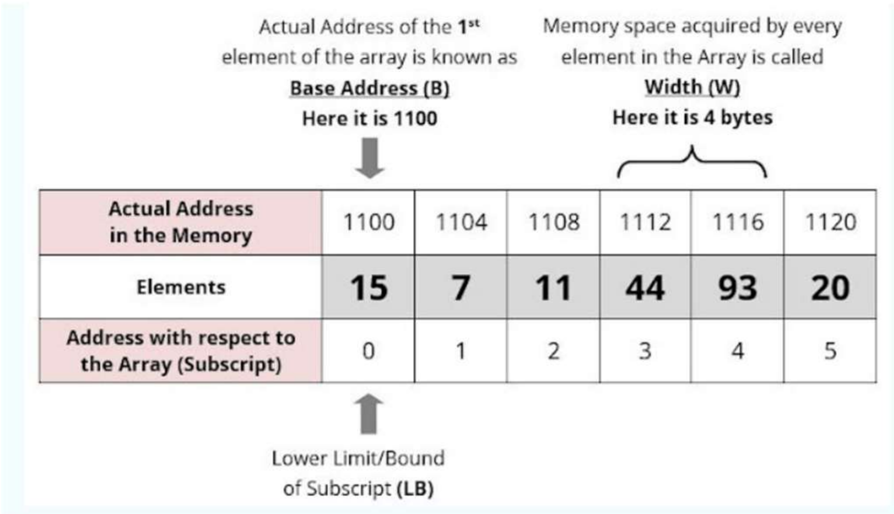


Address of A[i]? A + 4*I

Bound check? 0 <= i < 10

Benefit: the array may change dynamically

Address Calculation (one dimension Array)

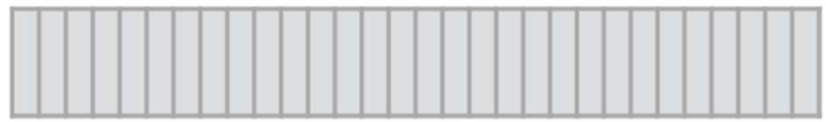


Address of A[3] = B + W * (i – LB)

= 1100 + 4 * (3 – 0) = 1112

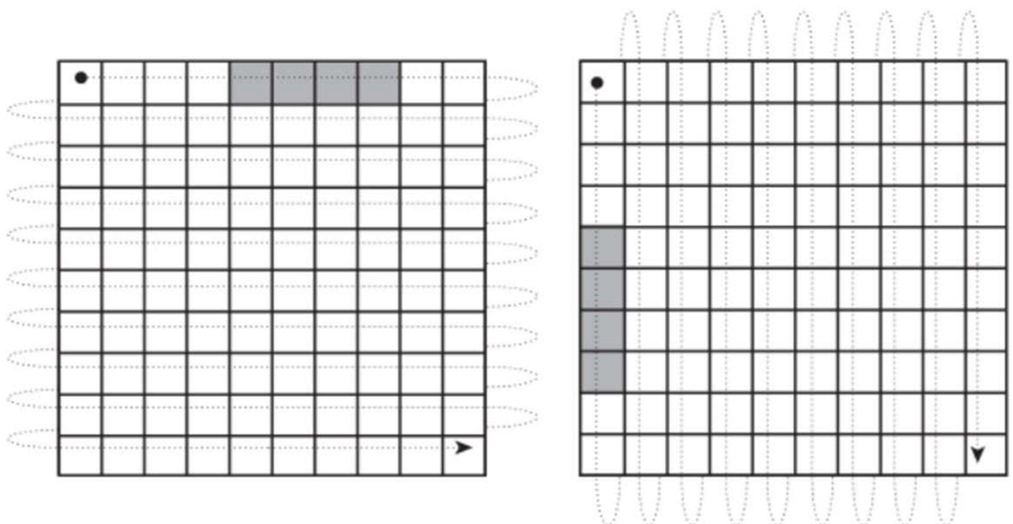
Memory Layout

One-dimensional arrays



Two-dimensional arrays

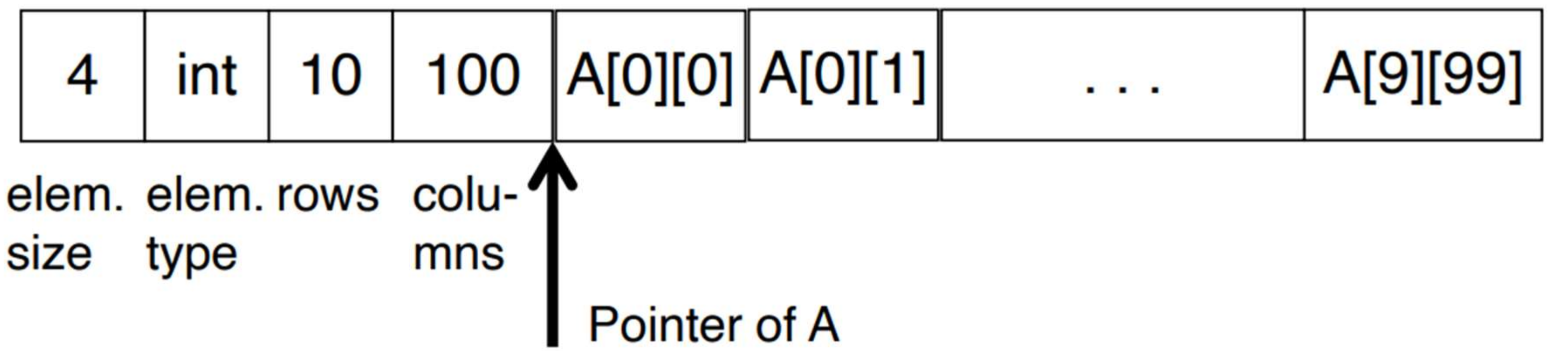
```
int[][] A = new int[10][100]
```



Row-major order Column-major order

7

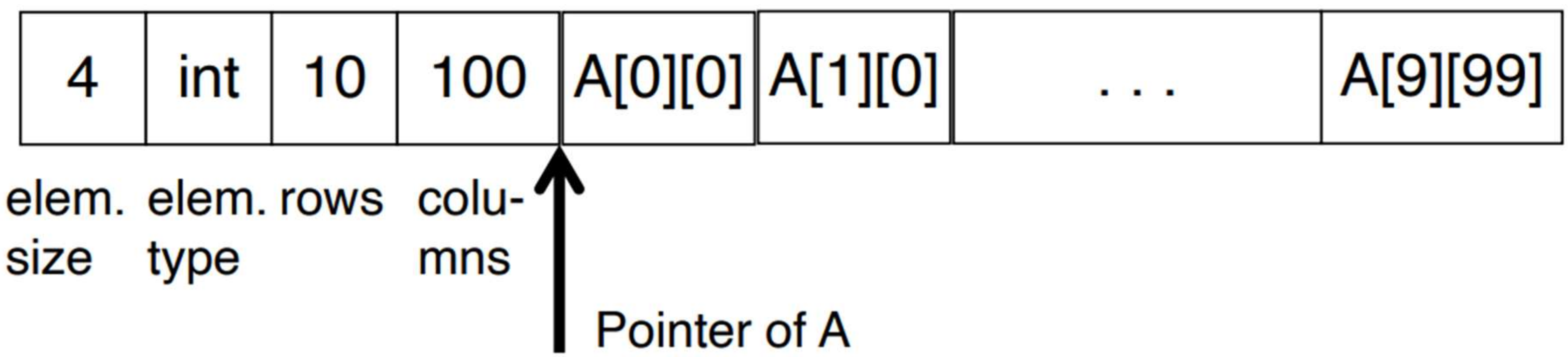
Address Calculation (Row Major)



Address of A[i][j]? A + 4(i*100+j)

Bound check? 0 <= i < 10, 0 <= j < 100

Address Calculation (Column Major)



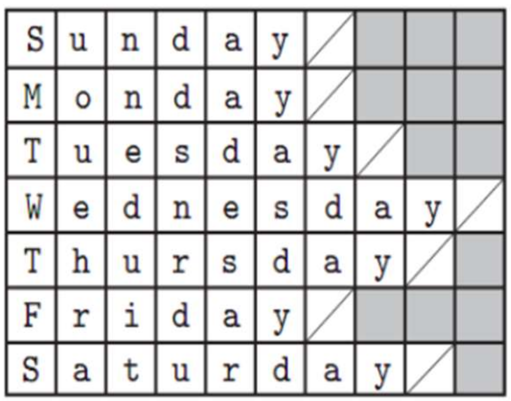
Address of A[i][j]? $A + 4(j*10+i)$

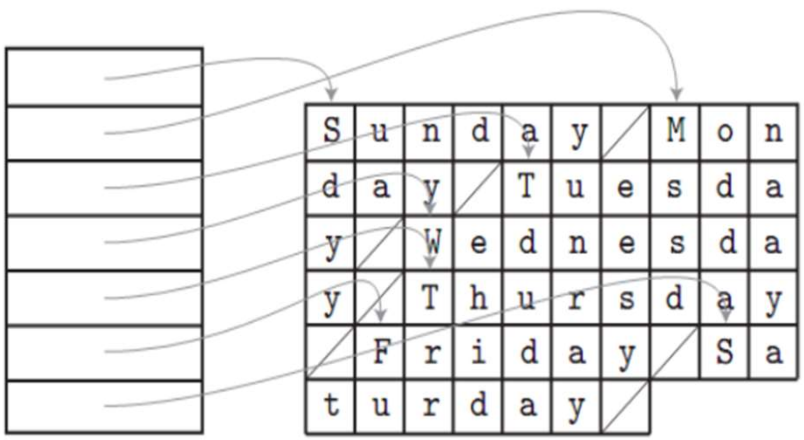
Bound check? $0 <= i < 10, 0 <= j < 100$

Memory Layout

Row-Pointer Layout

```
int[][] B = new int[10][]
B[0] = new int[100]
B[1] = new int[50]
```





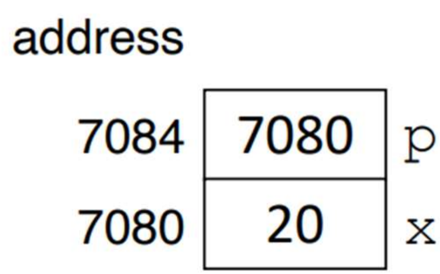
Pointers

What are they?

- A set memory addresses and operations on them

- Values: legal addresses, and a special value, nil

```
int x=20;
int* p = &x;
```



Pointers

Operations: assignment, dereferencing, arithmetic

```
int x=20;
int* p = &x;
int y=*p;
```

```
int a[3] = {1,2,3};
int x = *(a+1) //same as a[1]
```

Uses

- Indirect addressing (access arbitrary address)
- Manage dynamic storage (heap)

Pointers vs. References

- Pointers: `int *p;`
- References: `int &p;`

Value Model vs. Reference Model: A = B

- Value model: the value of B is copied to A
- Reference model: A is an alias of B (same memory)
- Java: primitive types follow value model; objects follow reference model

References

- Restricted pointers: cannot be used as value or operated in any way
- Not directly visible to the programmer No explicit data type

```
double r=2.3;
double& s=r;//s is an alias of r (share memory)
double *p = &r; //p has value: address of r
s += 1; *p += 1;
```

References

Uses

- ~~Indirect addressing (access arbitrary address)~~
- Manage dynamic storage (heap)

Alias of existing variable

The Nil Pointer Problem

“I call it my billion-dollar mistake. It was the invention of the null reference in 1965. … This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.”

C.A.R. Hoare, 2009

How to avoid it?

String

- Now so fundamental, directly supported.
- In C, a string is a 1D array with the string value terminated by a NUL character (value = 0).
- In Java, Perl, Python, a string variable can hold an unbounded number of characters
- Libraries of string operations and functions.

Function Type

- Pascal example:
  - function newton(a, b: real; function f: real): real;
  - Know that f returns a real value, but the arguments to f are unspecified.
- Strongly typed functional language
  - Functions are first class
  - Accepts functions as arguments and return functions as results

Function Type

- ML: T1 -> T2
  - -> is right-associative
    - T1 -> T2 -> T3 is the same as T1 -> (T2 -> T3)
  - Examples:
    - int -> int -> int
    - int -> (int -> int) -> int
    - (int -> int) -> (int -> int)

Subtypes

- S is a subtype of T if S extends or implements T
- A subtype is a type that has certain constraints placed on its values or operations.

Type Conversions

- A type conversion is usually done by a conversion function
  - e.g., 3 + 5.0; 3 + “hello”
- A *widening* conversion if the result type permits more bits
  - E.g., in C, byte b = 23; int i = (int) b;
- A type conversion is a *narrowing* conversion if the result type permits fewer bits, thus potentially losing information.
  - in C, int i = 1911; byte b = (byte) i;
- *implicit* narrowing conversions may be harmful
  - Why?