Types

CMPSC 461
Programming Language Concepts
Penn State University
Fall 2016

Kinds of Types

Primitive

Constructed

- Products
- Unions
- Arrays
- Lists

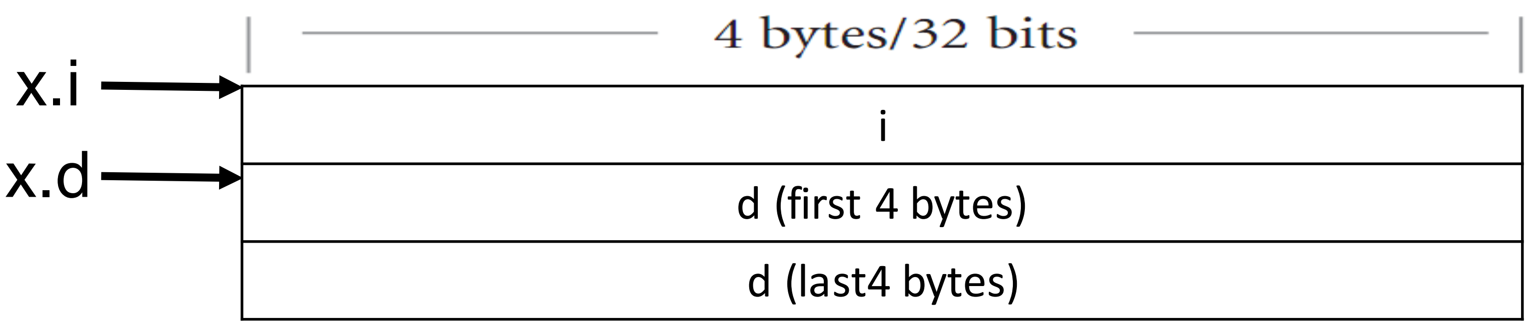
User-Defined

Records and Structures

Usually laid out contiguously

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

A possible memory layout:



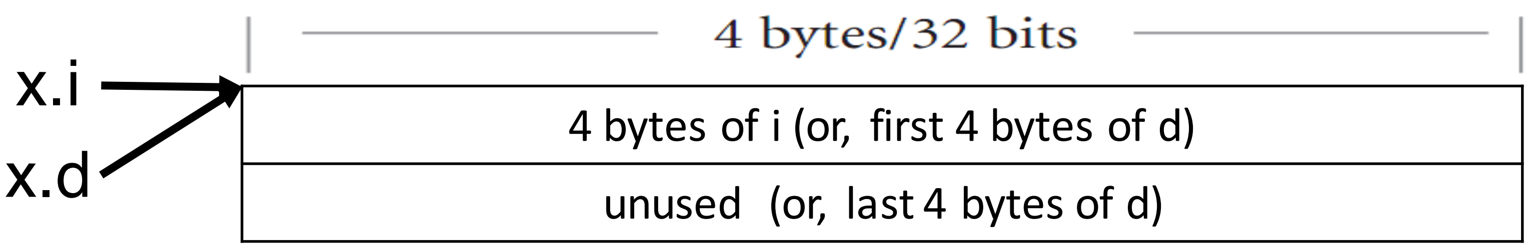
Each field has a separate piece of memory

(Free) 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

(Free) Union Types

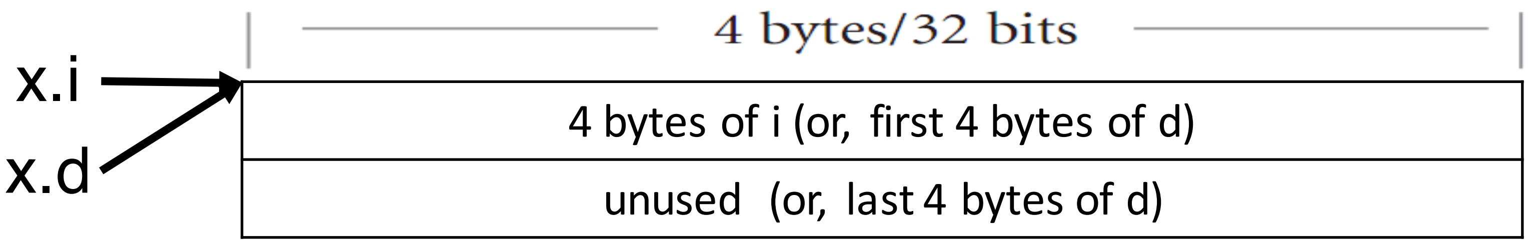
Not type safe:

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;
}
float x = 1.0;
if (u.which == isInt) u.a = 1;
if (u.which == isFloat) x = x + u.p;
```

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

Sum Types

Many functional programming languages support type-safe sum types

Tag

(possibly empty) payload type

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}
```

Analogy:

Values of color

Values of shape

Values of coloredShape

(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];
  … }
```

Array as Parameters & Conformant Arrays

```
int f(int A[], int size) {
  …
}
```

```
int f(int A[] ) {
  … A.length …
}
```

```
int f(int n, int M[n][n]) {
  …
}
```

Bounds Checking

Is it done?

How is it done?

When is it done?

```
int f() {
  int A[10];
  ...
  A[e]
  ...
}
```

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

C Array

Static/Stack/Heap allocated

Size statically/dynamically determined

Array bounds not checked (buffer overflow)

Java Array

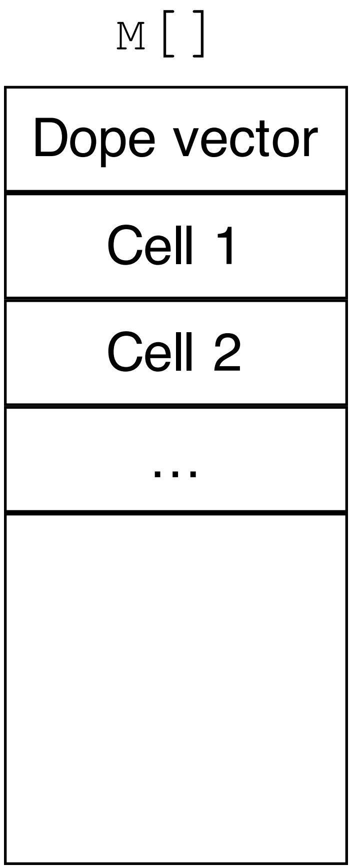
Heap allocated

Size dynamically determined

Array size is part of stored data (Dope Vector)

Array bounds checked

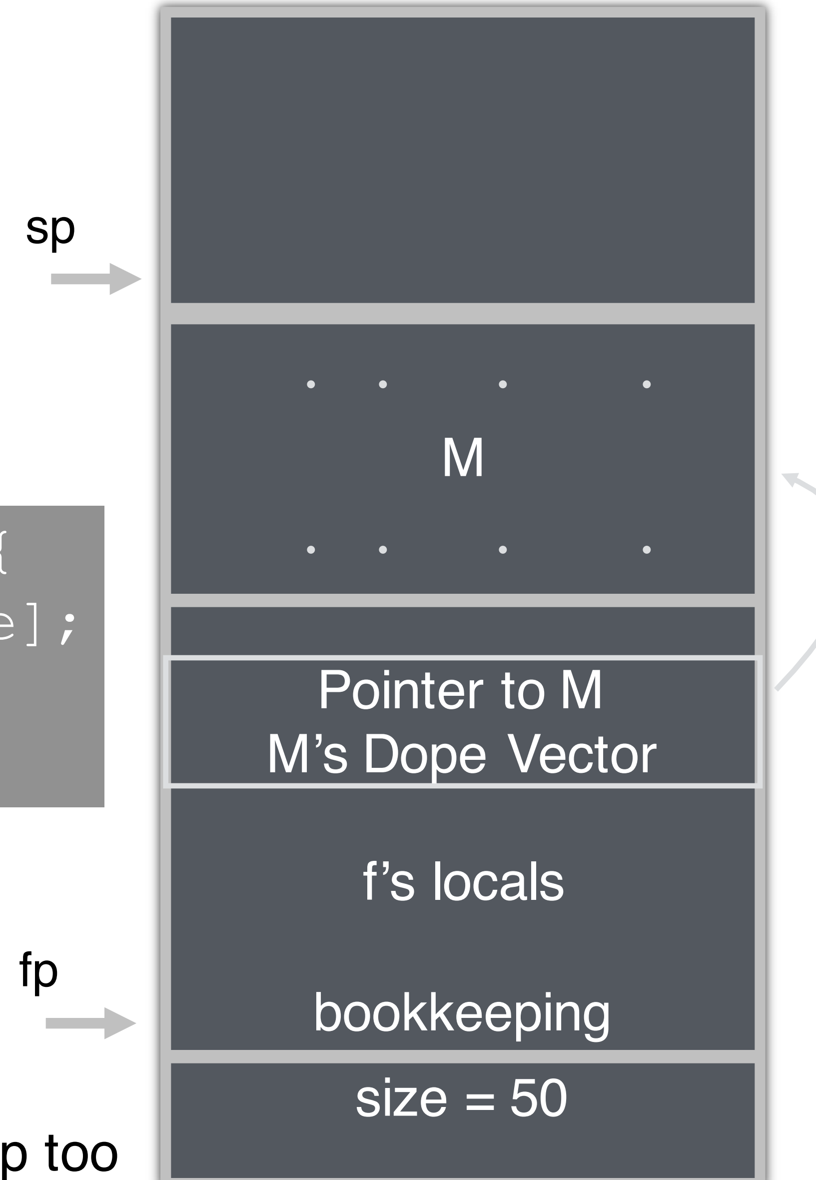
Dope Vectors



Conceptual view

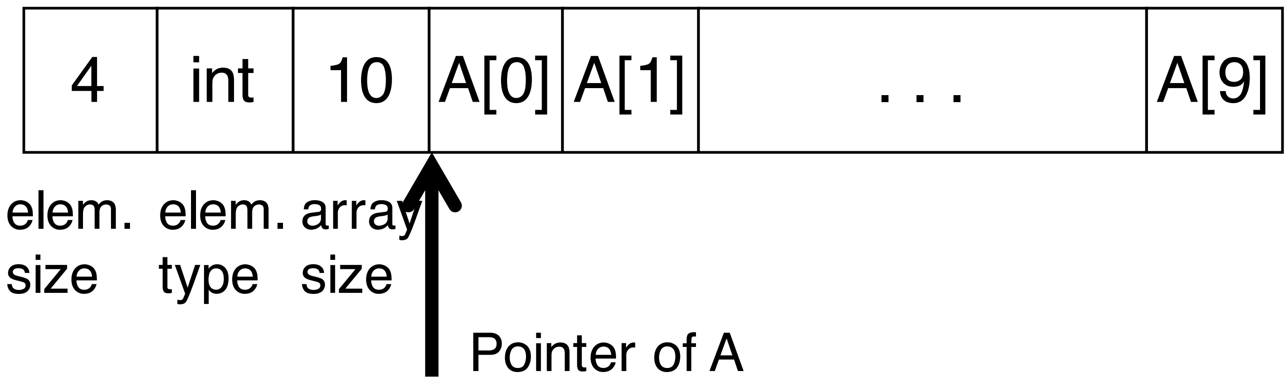
```
void f(int size) {
  int M[size][size];
  ...
}
```

Real view (when payload is allocated on stack)



The payload can be allocated on the heap too

Dope Vectors (one example)



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

Bound check? 0 <= i < 10

Benefit: the array may change dynamically