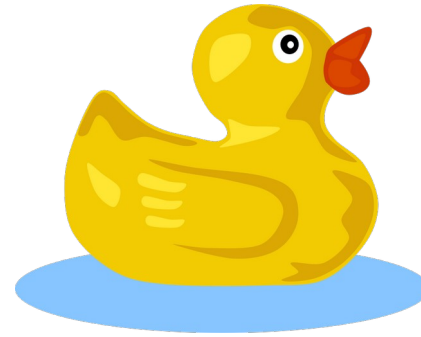


CS 430

Spring 2020

Mike Lam, Professor



$$\text{TEq} \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 \text{ '==' } e_2 : \mathbf{bool}}$$

Types and Checking

Type Systems

- **Type system**
 - Rules about valid types, type compatibility, and how data values can be used
- **Benefits of a robust type system**
 - Earlier error detection
 - Better documentation
 - Increased modularization

Data Types

- **Data type**: collection of values and associated operations
 - **Descriptor**: collection of a variable's attributes, including its type
- Primitive data types
 - Integer, floating-point, complex, decimal, boolean, character
- User-defined data types
 - Structured: arrays, tuples, maps, records, unions
 - Ordinal: enumerations, subranges
 - Pointers and references

Data Types

- **Primitive** data types
 - Integer: signed vs. unsigned, two's complement, arbitrary sizes
 - Tradeoff: storage/speed vs. range
 - Floating-point: IEEE standard (sign bit, exponent, significand), precision, rounding error
 - Tradeoff: storage/speed vs. accuracy, precision vs. range
 - Character: ASCII, Unicode, UTF-8, and UTF-16 (variable-length), UTF-32 (fixed-length)
 - Tradeoff: expressivity vs. storage space and processing time
 - Boolean: 0 (false) or 1 (true); usually byte-sized
 - Complex: pairs of floats (real and imaginary)
 - Decimal: binary coded decimal (4 bits per digit)

User-Defined Data Types

- **Structured**
 - Arrays and lists: indexed sequences of elements
 - Tuples: fixed-length sequence of elements
 - Associative arrays: mapping from keys to values (often w/ hashing)
 - Records: (name, type) pairs, dot notation, a.k.a. "structs"
 - Unions: different types at runtime, tag/discriminant, safety issues
- **Ordinal** (value \Leftrightarrow integer mapping)
 - Booleans and characters
 - Enumerations: subset of constants
 - Subranges: contiguous subsequence of another ordinal type

Data Types

- **Product** vs. **sum** types
 - Product types: cross product of other types
 - Like a struct in C (“*both/and*”)
 - Sum types: union of other types
 - Like a union in C (“*either/or*”)

```
/* product type */  
struct {  
    float f;  
    int i;  
} x;
```

```
/* sum type */  
union {  
    float f;  
    int i;  
} x;
```

Arrays and Lists

- **Arrays**

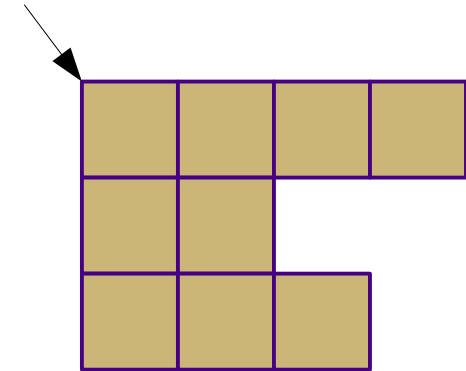
- Usually homogeneous (with fixed element width)
- Usually fixed-length
- Usually static or fixed stack/heap-dynamic
- Calculating index offsets: $\text{base} + \text{index} * (\text{element_size})$

- **Lists**

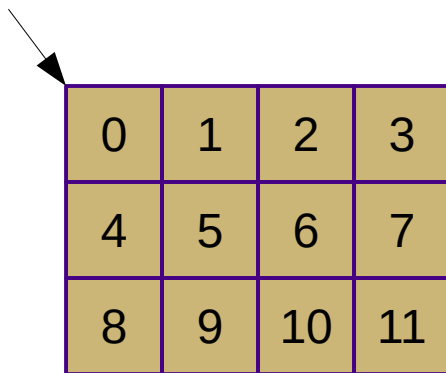
- Sometimes heterogeneous
- Usually variable-length
- Usually stack-dynamic or heap-dynamic
- In functional languages: usually defined as `head:tail`

Multidimensional Arrays

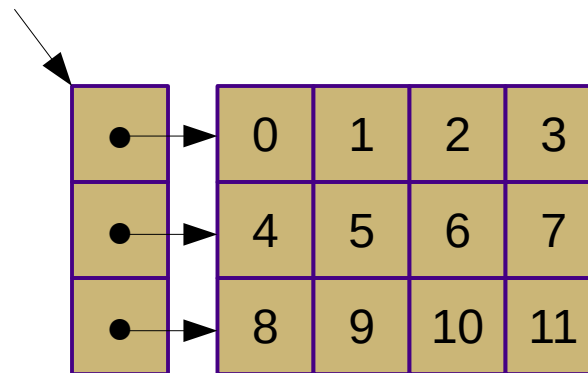
- **Multidimensional arrays**
 - True multidimensional vs. array-of-arrays
 - Row-major vs. column-major
 - Rectangular vs. jagged
 - Calculating index offsets



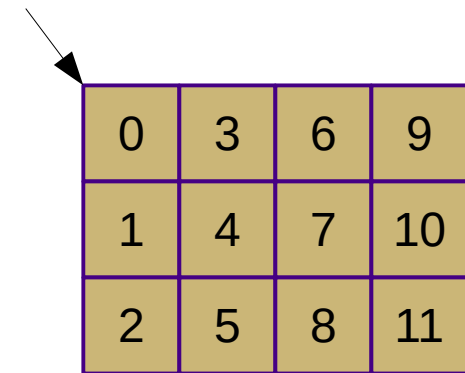
Ragged



Row-major



Row-major array-of-arrays



Column-major

Character Strings

- **Strings** are often stored as arrays of characters
- Common operations: length calculation, concatenation, slicing, pattern matching
- Questions:
 - Should the language provide special support?
 - Should string length be static or dynamic?
 - How should the length be tracked?
 - Should strings be immutable?
- Tradeoffs: speed vs. convenience
- Buffer/length overruns are a common source of security vulnerabilities

Subtypes

- A **subtype** is a constrained version of an existing type
 - Values of the subtype can often be used in place of the original, but not vice versa
 - E.g., in Ada: `subtype Small_Int is Integer range 0..100;`

Pointers and References

- **Pointer**: memory address or **null / nil / 0**
 - Adds a layer of indirection to memory accesses
 - Simplifies creation of dynamic data structures (e.g., trees)
 - Example of a **nullable** type
- **Reference**: refers to an object or value in memory
 - Different semantics than pointers (strictly safer)
- Design issues
 - Language support (pointers, references, or both?)
 - Scope and lifetime of pointer/reference and associated value
 - Type restrictions (must match? `void*` allowed?)
 - Arithmetic (generally allowed for pointers, not references)

Pointers

```
int x = 1;  
int y[4] = {2, 3, 4, 5};  
int *p = &x;  
*p = 6;  
p = y;  
*p = 7;
```

**What are the values of x and y
at the end?**

Pointers

```
int x = 1;
```

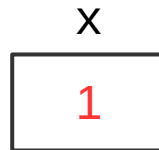
```
int y[4] = {2, 3, 4, 5};
```

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

```
*p = 6;
```

```
p = y;
```

```
*p = 7;
```



Pointers

```
int x = 1;
```

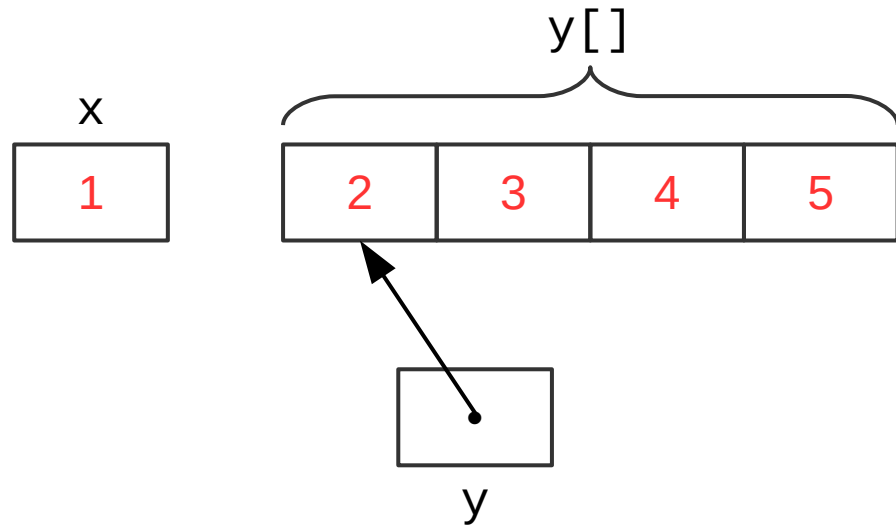
```
int y[4] = {2, 3, 4, 5};
```

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

```
*p = 6;
```

```
p = y;
```

```
*p = 7;
```



Pointers

```
int x = 1;
```

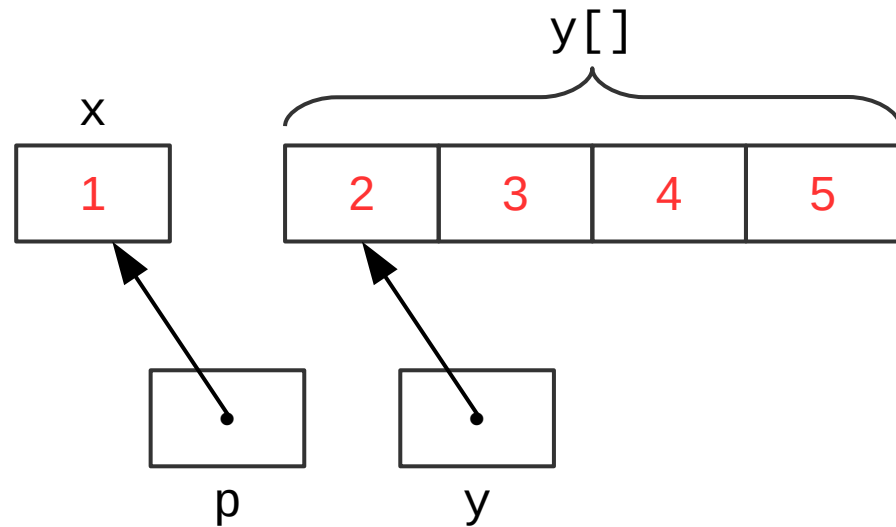
```
int y[4] = {2, 3, 4, 5};
```

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

```
*p = 6;
```

```
p = y;
```

```
*p = 7;
```



Pointers

```
int x = 1;
```

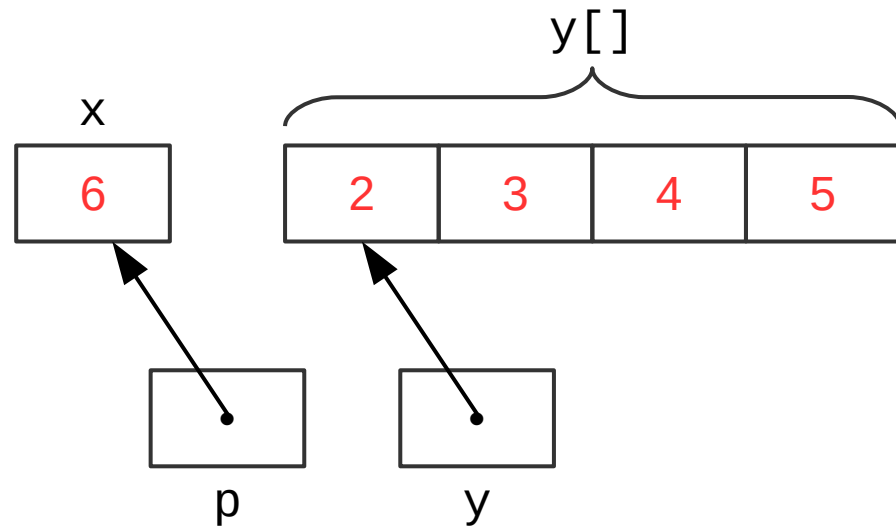
```
int y[4] = {2, 3, 4, 5};
```

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

```
*p = 6;
```

```
p = y;
```

```
*p = 7;
```



Pointers

```
int x = 1;
```

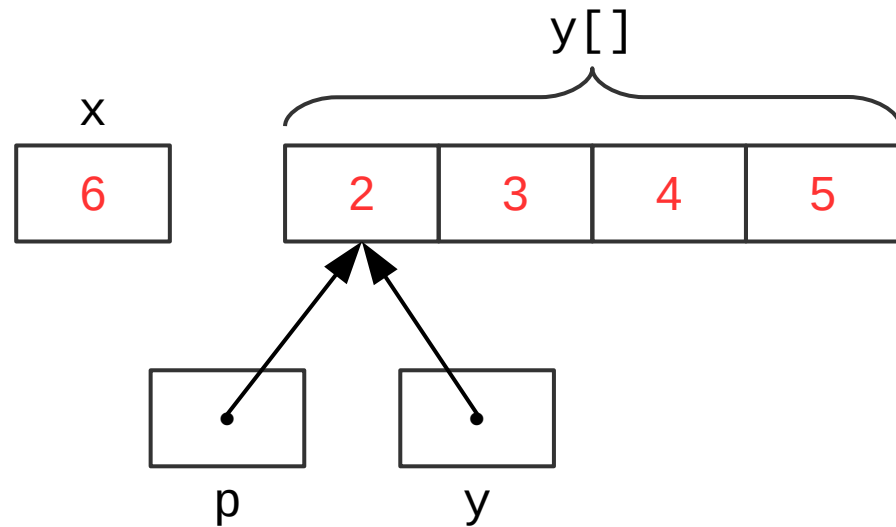
```
int y[4] = {2, 3, 4, 5};
```

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

```
*p = 6;
```

```
p = y;
```

```
*p = 7;
```



Pointers

```
int x = 1;
```

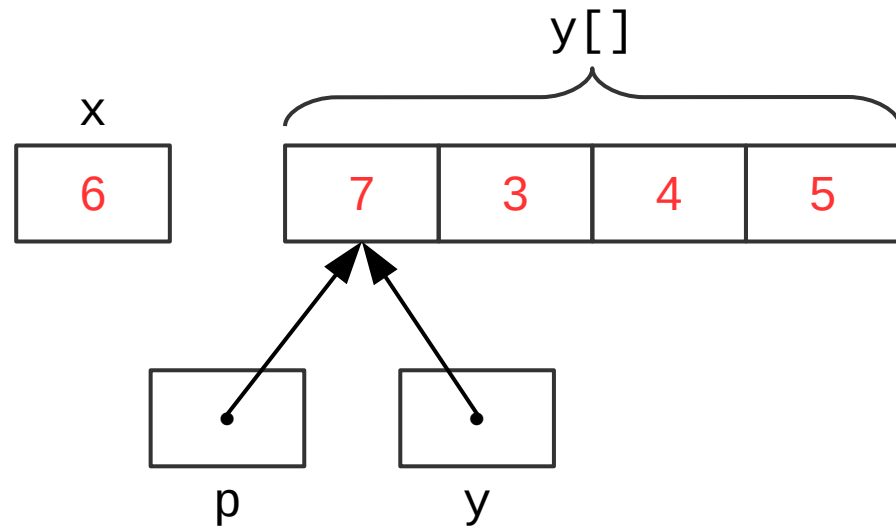
```
int y[4] = {2, 3, 4, 5};
```

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

```
*p = 6;
```

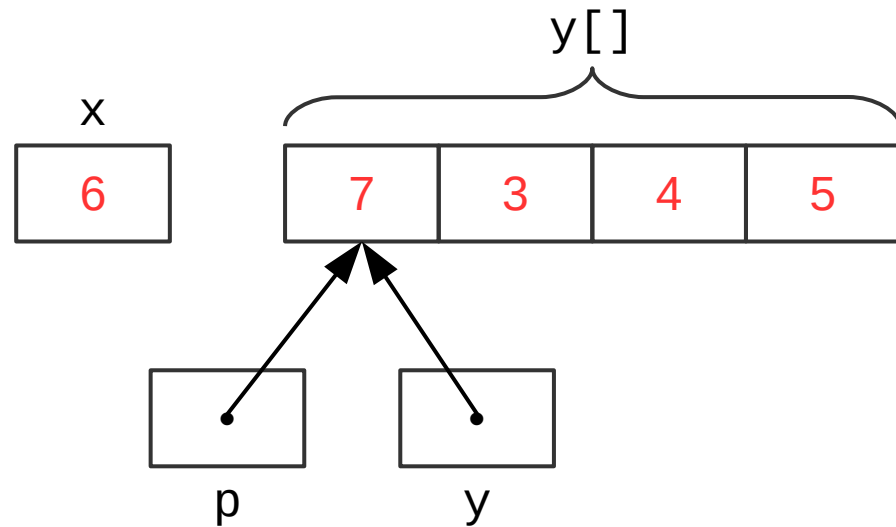
```
p = y;
```

```
*p = 7;
```



Pointers

```
int x = 1;  
int y[4] = {2, 3, 4, 5};  
int *p = &x;  
*p = 6;  
p = y;  
*p = 7;
```



What about this?

```
p++;  
*p = 9;
```

Pointer/Reference Issues

- **Dangling pointer**: variable is deallocated but pointer remains
 - Dereferencing pointer is invalid (might segfault; might not!)
 - Debuggers (e.g., gdb) can help
- **Memory leaks**: variable is still allocated but no longer accessible
 - In CPL: **lost heap-dynamic variables**
 - Memory remains allocated
 - Analysis tools (e.g., Valgrind/Memcheck) can help

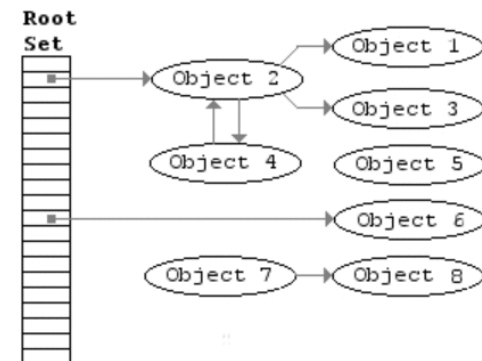
Dangling Pointer Solutions

- **Tombstones**
 - Extra level of indirection: new access pointer for each object
 - External pointers only point to tombstones
 - When deallocated, tombstone is set to null
 - Causes null pointer dereference if ever used
- **Locks and keys**
 - Pointers are stored as (key, address) pairs
 - Heap variables store key field as well
 - Pointer key and value key are compared for every reference
 - If the keys do not match, the access is invalid

Downside: high overhead!

Memory Leak Solutions

- **Reference counters**
 - Track # of references to an object
 - Deallocate object when counter hits zero
- **Mark-and-sweep**
 - Pause the application (sometimes unnecessary)
 - Initialize indicators for all memory cells to "unmarked"
 - Mark reachable heap memory cells by recursively following pointers from stack and static memory
 - Deallocate unmarked cells
 - Improvements:
 - Generational collection
 - Incremental collection



Type Checking

- **Type system**
 - Rules about how data values can be used
- **Type checking**
 - Act of ensuring that the type system is adhered to
 - Ensure that operands are of **compatible** types
 - Or of **equivalent** types if coercions aren't allowed
 - Violations are called **type errors**
 - Usually, type errors are considered to be bugs
 - Sometimes are reported only as warnings

Type Checking

- Language design decisions:
 - Are declarations explicit or implicit?
 - Which types are equivalent?
 - Are type conversions allowed?
 - Can multiple types be used in some places?
 - When does type checking occur?
 - In general, how pedantic is the process?

Type Checking

- **Type declarations**
 - **Explicit**: types required
 - E.g., `int x = 5; float y = 4.2;`
 - **Implicit**: types not required (or even not allowed)
 - E.g., `x = 5; y = 4.2;`
 - Types are bound at assignment
 - However, these types can often be **inferred** statically
 - Tradeoff: readability vs. writability and expressiveness

Type Checking

- **Type equivalence: name vs. structure**
 - **Named** types vs **anonymous** types
 - **Aliased** types (e.g., typedef in C)
 - Examples:

```
typedef float celsius;  
typedef float fahrenheit;
```

```
celsius a = 25.7f;  
fahrenheit b;
```

```
b = a;    OK in C
```

```
typedef struct { int x; } box;  
typedef struct { int x; } bin;
```

```
box c;  
c.x = 5;
```

```
bin d;  
d = c;    NOT OK in C
```

```
struct { int x; } e;
```

```
e = c;    NOT OK in C
```

C permits structure equivalence except for records, where it requires name equivalence

Type Compatibility

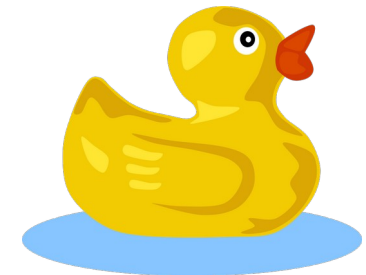
- **Type conversions**
 - If variables of a particular type can be converted to a different type, those types are **compatible**
 - **Widening** vs. **narrowing**
 - Latter may cause information loss
 - **Implicit** vs. **explicit**
 - Implicit: **coercion**, e.g., `float x = 5;`
 - Explicit: **casting**, e.g., `int x = (int)3.14;`

Type Polymorphism

- Object-oriented inheritance
 - Example of **subtypes**
- Parameterized functions
 - Uses generic **type variables**
 - Example: generic list functions in Haskell
 - E.g., `head : [a] → a` `length : [a] → int`
- Abstract data types
 - Models of generic data structure behavior
 - Implementation is hidden from user
 - Can use **parameterized** types
 - E.g., a `queue<float>` or `queue<int>`
 - Examples: C++ templates and Java generics

Type Checking

- Static vs. dynamic type checking
 - **Static** type checking finds type errors at compile time
 - Usually checked by compiler (e.g., Haskell)
 - **Dynamic** type checking find types errors at run time
 - Usually checked by runtime system (e.g., Ruby, Python)
 - “**Duck typing**” is a particular form of dynamic typing
 - If an object has a method, you can call it! (“if it quacks like a duck...”)
 - **Hybrid**: some static, some dynamic
 - E.g., C++, Java
 - Tradeoff: overhead vs. flexibility



Type Checking

- **Type rules** are sometimes expressed using proof notation
 - Premises on top, conclusion at the bottom

$$\text{TDec} \frac{}{\vdash \text{DEC} : \text{int}}$$

“all decimal literals are integers”

$$\text{TTrue} \frac{}{\vdash \text{true} : \text{bool}}$$

“the ‘true’ literal is a boolean”

$$\text{TAdd} \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 \text{ '+' } e_2 : \text{int}}$$

“the result of an addition is an int, as long as both operands are ints”

$$\text{TEq} \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 \text{ '==' } e_2 : \text{bool}}$$

“the result of an equality check is a boolean, as long as both operands are of the same type”

$$\text{TFuncCall} \frac{\text{ID} : (\tau_1, \tau_2, \dots, \tau_n) \rightarrow \tau_r \in \Gamma \quad \Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \dots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash \text{ID} \text{ '(' } e_1, e_2, \dots, e_n \text{ ')'} : \tau_r}$$

“the result of a function call is its return type, as long as all actual parameters match the types of their corresponding formal parameters”

Type Checking

- **Strong** vs. **weak** typing
 - Strong typing: all type errors are detected
 - Tradeoff: safety vs. expressiveness
 - Terms often used somewhat loosely
- Evidence of strong typing
 - Static type checking
 - Type inference (even for implicit typing!)
- Evidence of weak typing
 - Type coercions (manual or automatic)
 - Pointer or union types