Data Types

COS 301 - Programming Languages

Fall 2018

Types

- *Type* – collection of values + operations on them
- Ex: integers:
  - values: …, -2, -1, 0, 1, 2, …
  - operations: +, -, *, /, <, >, …
- Ex: Boolean:
  - values: true, false
  - operations: and, or, not, …

Bit Strings

- Computer: Only deals with *bit strings*
- No intrinsic "type"
- E.g.:

  0100 0000 0101 1000 0000 0000 0000 0000

  could be:
  - The floating point number 3.375
  - The 32-bit integer 1,079,508,992
  - Two 16-bit integers 16472 and 0
  - Four ASCII characters: @ X NUL NUL
- What else?
- What about 1111 1111?

Levels of Abstraction

- First: machine language, bit strings
- Then: assembly language
  - Mnemonics for operations, but also…
  - ...human-readable representations of bit strings
- Then: HLLs
  - Virtual machine – hides real machine's registers, operations, memory
  - Abstractions of data: maps human-friendly abstractions ⇒ bit strings
  - Sophisticated typing schemes for numbers, characters, strings, collections of data, …
  - OO – just another typing abstraction

Types in Early Languages

- Early languages: types built in (FORTRAN, ALGOL, COBOL)
- Suppose you needed to represent colors
  - Map to integers
  - But:
    - carry baggage of integer operations (what does it mean to multiply two colors?)
    - no type-specific operations (blending, e.g.)
  - E.g., days of the week, cards in a deck, etc.

Evolution

- FORTRAN:
  - integers, “reals”, complex, character (string), logical
  - arrays as structured type
- Lisp:
  - Symbols, linked lists, integers, floats (later rationals, complex, arrays,…)
- COBOL:
  - programmer could specify accuracy
  - records

Evolution

- Algol 68:
  - few basic types
  - structure defining mechanisms (user defined types)
- 1980's: abstract data types (**ADTs**)
- Abstract data types ⇒ objects (though first developed in 1960's)

Type Errors

- **Type error**:
  - operation attempted on data type for which it is undefined
  - operation could be just assignment
- Machine data carries no type information.
- Assembly language:
  - type errors easy to make,
  - little if any type checking
- HLLs ⇒ reduce type errors
  - Greater abstraction ⇒ fewer type errors
  - Type system: type checking, detecting type errors

Data types: Issues

- How to associate types with variables?
  - Recall **symbol table:** info about all variables
  - Descriptor in symbol table: all attributes
- What operations are defined?
- How are they specified?
- Implementation of types?

Overview

- Primitive data types
- Character strings
- User-defined ordinal types
- Arrays
- Associative arrays
- Records
- Unions
- Pointers & references
- Miscellaneous types
- Type equivalence
- Functions as types
- Heap management

Primitive Data types

Primitive data types

- **Primitive data type:**
  - not defined in terms of others (scalar) or…
  - …provided natively by language (e.g., strings, arrays sometimes)
- Some very close to hardware: integers, floats
- Others: require non-hardware support

Primitive scalar data types:

Type	C	Ada	Java	Python	Lisp
Byte	char	none	byte	none	none (bit-vector)
Integer	short, int, long	Integer, Natural, Positive	short, int, long	int	fixnum, bignum,
Float	float, double, ext'd double	Float, Decimal	float, double	real	single-float, double-float, ratio
Char	char	Character	char	none (string)	character
Bool	none (0, not zero)	Boolean	boolean	bool	nil, t (and anything not nil)

Integers

- Generally direct mapping to machine representation
- Most common:
  - **sign-magnitude**
  - **two's complement**
- Others:
  - Unsigned (binary)
  - Binary coded decimal

Review: Sign-magnitude

- Binary number, high-order bit is **sign bit**
- E.g.: -34 in 8 bits:
  - binary 34 → 0010 0010
  - sign-magnitude -34 → 1010 0010
- Easy, but:
  - 2 representations of 0
  - have to treat high-order bit differently

Review: 2's complement

- Divide possible range of n-bit binary numbers:
  - $0 - 2^{n-1}-1$ ⇒ positive numbers
  - 2^{n-1} to $2^{n}-1$ ⇒ negative numbers
  - E.g., 8 bits:
    - Positive 1 = 0000 0001
    - Negative 1?
      - Odometer-like
      - 1111 1111
      - 1 + (-1) = 0: 0000 0001 + 1111 1111 = (1)0000 0000

Review: 2's complement

- Mechanics:
  - Take **1's complement**, add 1
  - E.g.: -34 in 2's complement
    - 34 = 0010 0010 in binary
    - 1's complement: 1101 1101
    - 1101 1101 + 1 ⇒ 2's complement: 1101 1110
- Advantages: subtraction can be done with addition

Review: 2's complement

- Example: 123 - 70 in 8 bits:
  - $123_{10} \Rightarrow 0111\ 1011_2$
  - $70_{10} \Rightarrow 0100\ 0110_2$
  - $-70_{10} \Rightarrow 1011\ 1001_2 + 1 = 1011\ 1010_2$

$$\begin{array}{r} 0111\ 1011 \\ +\ 1011\ 1010 \\ \hline (1)00110101 \\ \Rightarrow 53_{10} \end{array}$$

Size of integers

- Generally implementation-dependent
- E.g., C/C++:
    - signed and unsigned
    - `byte`, `short`, `int`, `long`
- Exception: Java
    - **`byte`** = 8 bits
    - **`short`** = 16
    - **`int`** = 32
    - **`long`** = 64
- Ada: programmer can specify size, error at compile time if too large

Fixed-size integers

- **Unsigned integers**: e.g. C/C++
  - Why?
- Problem: how to mix operations?

```
unsigned char foo = 128;
int bar = 1;
int baz;
baz = foo + bar;
```

  - `foo` will be represented as 1000 0000
  - So will **baz** be 128+1 or -128+1? → may depend on implementation!
  - Safer — **casting**:

```
baz = (int)foo + bar;
```

Overflow

- When can it occur?
  - Unsigned, sign-magnitude ⇒ result larger than representation can handle
  - Two's-complement representation ⇒ **wraparound**
- Many languages do not generate overflow exception — Why not?

Arbitrary-precision integers

- **Fixed-length integers:** close mapping to hardware:
  - Pro: efficient
  - Con: limited range
- Conceptually-unlimited range: **arbitrary precision integers**
  - Started with Lisp's `bignum` type
  - Other languages: Ruby, Python, Haskell, Smalltalk
- Requires software support ⇒ not as efficient
- Limited only by available memory
- May start with small (machine-based) integer, switch as numbers get too large

Arbitrary-precision integers

- E.g., in Lisp, Fibonacci(10000) =

3364476487643178326662161200510754331030214846068006390656476997468008144216666236815559551363373402
5582065332680836159373734790483865268263040892463056431887354544369559827491606602099884183933864652
7313000888302692356736131351175792974378544137521305205043477016022647583189065278908551543661595829
8727968298751063120057542878345321551510387081829896979161312785626503319548714021428753269818796204
6936097879900350962302291026368131493195275630227837628441540360584402572114334961180023091208287046
0889239623288354615057765832712525460935911282039252853934346209042452489294039017062338889910858410
6518317336043747073790855263176432573399371287193758774689747992630583706574283016163740896917842637
8624212835258112820516370298089332099905707920064367426202389783111470054074998459250360633560933883
8319233867830561364353518921332797329081337326426526339897639227234078829281779535805709936910491754
7080893184105614632233821746563732124822638309210329770164805472624384237486241145309381220656491403
2751086643394517512161526545361333111314042436854805106765843493523836959653428071768775328348234345
5573667197313927462736291082106792807847180353291311767789246590899386354593278945237776744061922403
3763867400402133034329749690202832814593341882681768389307200363479562311710310129195316979460763273
7589253530772552375943788434504067715555779056450443016640119462580972216729758615026968443146952034
6149322911059706762432685159928347098912847067408620085871350162603120719031720860940812983215810772
8207635318662461127824553720853236530577595643007251774431505153960090516860322034916322264088524885
2433158051534849622434848299380905070483482449327453732624567755879089187190803662058009594743150052
4025327097469953187707243768259074199396322659841474981936092852239450397071654431564213281576889080
5878318340491743455627052022356484649519611246026831397097506938264870661326450766507461151267752274
8621598642530711298441182622661057163515069260029861704945425047491378115154139941550671256271197133

252763631939606902895650288268608362241082050562430701794976171121233066073310059947366875

- $= 10^{2089}$
- This is the **only** way to represent this number — (much, much) larger than a double float type!

Floating point numbers

- *Not* = real numbers – only **some** real numbers
- Limited exponents ⇒ rules out very large, very small reals
- Irrational numbers cannot be represented (duh)
- Can't represent repeating rationals
  - These may not be what you think!
  - ⅓ in binary is repeating…
  - ...but so is 0.1!
- Limited precision ⇒ can't represent some **non-repeating** rational numbers

Floating point type

- Usually at least two floating point types supported (e.g., `float`, `double`)
- Usually exactly reflects hardware
  - Currently: IEEE Floating-Point Standard 754
  - Some older data was in different format
    - Can't precisely be represented in new format
    - So only accessible via software emulation of old hardware

IEEE floats

- Instead of decimal point, have a **binamal** *point* (or just **radix point** for general concept)
- Only two digits in binary (duh again)
  - **Normalize** number so that there is a 1 in front of the binamal point
  - E.g.: $0.0001010 \Longrightarrow 1.010 \times 2^{-4}$
  - But since all numbers (except 0) start with 1 $\Longrightarrow$ don't store the 1 — "hidden bit"
  - **Significand:** fractional part

IEEE floats

- **Exponent** is bias 127 – subtract 127 from it to get actual exponent
- Number = $(-1)^S \times 1.F_2 \times 2^{(E-127)}$

  where S is sign (0=pos, 1=neg), F is significand, and E is exponent (that is stored)

- Example: sign bit, 8-bit exponent, 23-bit unsigned fraction:

  0 0001 0000 0100 0000 0000 0000 0000 000 ⟹

  $(-1)^0 \times 1.01_2 \times 2^{(16-127)} = 1.25 \times 2^{-111}$

  $= 4.814824861 \times 10^{-34}$

IEEE floats: 0, NaN…

- Potential problem:
  - Any power of two: $1.0 \times 2^n \Rightarrow (0)^S \times 1.00 \times 2^{([127+n]-127)}$
  - $2.0 = 1.0 \times 2^1 \Rightarrow (0)^S \times 1.0 \times 2^{(128-127)}$
  - $1.0 = 1.0 \times 2^0 \Rightarrow (0)^S \times 1.00 \times 2^{(127-127)}$

    0 0000 0000 0000 0000 0000 0000 0000 000
  - How can you tell this from 0?
  - Alternatively, how would you even represent 0 in this notation?
  - 0 0000 000 0000 0000 0000 0000 0000 0
- **NaN** (not a number): S = 0/1, F = non-zero, E = all 1s
- +/- infinity: S = 0/1, F = zero, E = all 1s

IEEE floats: 0, NaN…

- Solution: define

  0 0000 0000 0000 0000 0000 0000 0000 000

  to be zero: S=0, E=0, F=0

- Some languages allow other “numbers”:
  - NaN (not a number): S = 0/1, F = non-zero, E = all 1s
  - +/- infinity: S = 0/1, E = all 1s, F = 0

IEEE 64-bit floats (double)

- Range for float (32 bits): approx. $\pm 10^{38}$ with 6-7 digits of precision
- Double ⇒ 64 bits; range approx. $\pm 10^{308}$ with 14-15 digits of precision
  - Sign bit + 11-bit exponent (bias-1023) + 52-bit unsigned fraction
  - $\text{Val} = (-1)^S \times 1.F_2 \times 2^{(E-1023)}$

IEEE floats

- How would you represent the following as an IEEE 32-bit float?
    - -2048.328125

IEEE floats

- How would you represent the following as an IEEE 32-bit float?
  - -2048.328125
    - 2048 in binary = 1000 0000 0000
    - 0.328125 = 1/4 + 1/16 + 1/64, in binary = 0.010101
    - So 2048.328125 = 1000 0000 0000.0101 01
    - Normalized = $1.00000000000010101 \times 2^{11}$
    - number = $(-1)^S \times 1.F_2 \times 2^{(E-127)}$
    - S = 1, F = 00000000000010101, E = 138 = $1000\ 1010_2$
    - Representation = 1 100 0101 0000 0000 0000 0101 0100 0000

Rational numbers

- Some languages provide **rational numbers** directly
- E.g., Lisp's "ratio" data type, Haskell's "Rational" data type
- Stores numerator and denominator as integers — usually reduced, i.e., with no common divisor > 1
- Arithmetic done specially
- Advantages: eliminates floating point errors

Rational numbers

- E.g.,

```
CL-USER> (loop for i from 1 to 1000
                    sum (/ 1 3.0))
333.3341
CL-USER> (loop for i from 1 to 1000
                    sum 1/3)
1000/3
CL-USER> (float (loop for i from 1 to 1000
                           sum 1/3))
333.33334
```

Complex numbers

- Some languages support **complex numbers** as primitive type
- E.g., Lisp, C (99+), Fortran, Python
- Represented as two floats (real & imaginary parts)
- E.g.:
  - Python: `(7 + 3j)`
  - Lisp: `#C(1 1)`

Decimal type

- Useful for business — COBOL, also C#, DBMS
- Stores fixed number of decimal digits
  - Usually **binary coded decimal (BCD)**

    E.g. 2758 $\Longrightarrow$ 0010 0111 0101 1000

  - Some languages: ASCII
- Some hardware: direct support
- Pro: accuracy – exact decimal precision (within reason)
- Cons: Limited range, more memory, slightly inefficient storage, & requires more CPU time for computation (unless hardware support)

Boolean type

- Two values
- Advantage: readability
- Could be bits, but usually bytes (**smallest addressable unit**)
- Some languages lack this type – C pre-1999, e.g.
- When no Boolean type, usually use integers: 0 = false, non-zero = true
- Other languages:
  - Perl – `false: 0, '0', '', (), undef`
  - Python – false: `None, False, 0, '', (), [], {}`, some others
  - Lisp – false = `nil`, otherwise true (including `t`)
  - PHP – false = `""`, true = `1` (also `FALSE`, `TRUE`)

Characters

- **Characters**: coded as bit strings (numbers)
- **ASCII**
  - American Standard Code for Information Interchange
  - Early and long-standing standard
  - 7-bit code originally; usually 8-bit now
- **EBCDIC**
  - Extended Binary Coded Decimal Interchange Code
  - IBM mainframes
  - 8-bit code

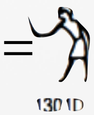
ASCII

- 7-bit code, but generally languages store as bytes (e.g., C's `char` type)
- The upper 128 characters – vary by OS, other software
- ISO 8859 encoding: uses the additional codes to encode European languages

Unicode

- As computer use (esp. the Web) became globalized ⇒ needed more characters
- Unicode designed to handle the ISO 10646 Universal Character Set (UCS)
- UCS: a 32-bit "alphabet" of all known human characters

Unicode

- Characters exist in Unicode as *code points* that then get mapped to representations
  - E.g., U+0045 = D, U+3423 = , U+1301D= 
  - Different encodings map these to different numeric codes
- Can encode all human languages
- Also: private use area – has been used to encode, e.g., Klingon

- Modern languages have adopted Unicode, including Java, XML, .NET, Python, Ruby, etc.
- Common codes: UTF-8, UTF-16, UTF-32

Unicode

- UTF8:
  - Most common on Web (> 90% of pages)
  - 1-4 byte code, can encode entire code point space
  - Byte 1: backward compatible w/ ASCII — encodes 128 characters

Number of bytes	Bits for code point	First code point	Last code point	Byte 1	Byte 2	Byte 3	Byte 4
1	7	U+0000	U+007F	0xxxxxxx			
2	11	U+0080	U+07FF	110xxxxx	10xxxxxx		
3	16	U+0800	U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	
4	21	U+10000	U+10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

Unicode

- Good introduction to Unicode:

The Absolute Minimum Every Software Developer Absolutely, Positively Must Know about Unicode and Character Sets (No Excuses!)

http://www.joelonsoftware.com/articles/Unicode.html

Character Strings


HAPPY HALLOWEEN!

Character String Types

- Strings: sequences of characters
- Design issues:
  - Primitive type? Or kind of array?
  - Length - static or dynamic?

Character String Operations

- Assignment, copying
- Comparison
- Concatenation
- Accessing a character
- Slicing/substring reference
- Pattern matching

String Libraries

- Some languages: not much support for string operations
- Most languages: string libraries
- Libraries for: primitive operations, regular expressions, substring replacement, etc.

Example: PHP string

- addcslashes — Quote string with slashes in a C style
- addslashes — Quote string with slashes
- bin2hex — Convert binary data into hexadecimal representation
- chop — Alias of rtrim
- chr — Return a specific character
- chunk_split — Split a string into smaller chunks
- convert_cyr_string — Convert from one Cyrillic character set to another
- convert_uudecode — Decode a uuencoded string
- convert_uuencode — Uuencode a string
- count_chars — Return information about characters used in a string
- crc32 — Calculates the crc32 polynomial of a string
- crypt — One-way string encryption (hashing)
- echo — Output one or more strings
- explode — Split a string by string
- fprintf — Write a formatted string to a stream
- get_html_translation_table — Returns the translation table used by htmlspecialchars and htmlentities
- hebrev — Convert logical Hebrew text to visual text
- hebrevc — Convert logical Hebrew text to visual text with newline conversion
- html_entity_decode — Convert all HTML entities to their applicable characters
- htmlentities — Convert all applicable characters to HTML entities

Example: PHP string

- html_entity_decode — Convert all HTML entities to their applicable characters
- htmlentities — Convert all applicable characters to HTML entities
- htmlspecialchars_decode — Convert special HTML entities back to characters
- htmlspecialchars — Convert special characters to HTML entities
- implode — Join array elements with a string
- join — Alias of implode
- lcfirst — Make a string's first character lowercase
- levenshtein — Calculate Levenshtein distance between two strings
- localeconv — Get numeric formatting information
- ltrim — Strip whitespace (or other characters) from the beginning of a string
- md5 — Calculate the md5 hash of a string
- metaphone — Calculate the metaphone key of a string
- money_format — Formats a number as a currency string
- nl_langinfo — Query language and locale information
- nl2br — Inserts HTML line breaks before all newlines in a string
- number_format — Format a number with grouped thousands
- ord — Return ASCII value of character
- parse_str — Parses the string into variables

Example: PHP string

- print — Output a string
- printf — Output a formatted string
- quoted_printable_decode — Convert a quoted-printable string to an 8 bit string
- quoted_printable_encode — Convert a 8 bit string to a quoted-printable string
- quotemeta — Quote meta characters
- rtrim — Strip whitespace (or other characters) from the end of a string
- setlocale — Set locale information
- sha1 — Calculate the sha1 hash of a string
- similar_text — Calculate the similarity between two strings
- soundex — Calculate the soundex key of a string
- sprintf — Return a formatted string
- sscanf — Parses input from a string according to a format
- str_getcsv — Parse a CSV string into an array
- str_ireplace — Case-insensitive version of str_replace.
- str_pad — Pad a string to a certain length with another string
- str_repeat — Repeat a string
- str_replace — Replace all occurrences of the search string with the replacement
- str_rot13 — Perform the rot13 transform on a string
- str_shuffle — Randomly shuffles a string

Example: PHP string

- str_split — Convert a string to an array
- str_word_count — Return information about words used in a string
- strcasecmp — Binary safe case-insensitive string comparison
- strchr — Alias of strstr
- strcmp — Binary safe string comparison
- strcoll — Locale based string comparison
- strcspn — Find length of initial segment not matching mask
- strip_tags — Strip HTML and PHP tags from a string
- stripcslashes — Un-quote string quoted with addcslashes
- stripos — Find position of first occurrence of a case-insensitive string
- stripslashes — Un-quotes a quoted string
- stristr — Case-insensitive strstr
- strlen — Get string length
- strnatcasecmp — Case insensitive string comparisons using a "natural order" algorithm
- strnatcmp — String comparisons using a "natural order" algorithm
- strncasecmp — Binary safe case-insensitive string comparison of the first n characters
- strncmp — Binary safe string comparison of the first n characters

Example: PHP string

- strpbrk — Search a string for any of a set of characters
- strpos — Find position of first occurrence of a string
- strrchr — Find the last occurrence of a character in a string
- strrev — Reverse a string
- strripos — Find position of last occurrence of a case-insensitive string in a string
- strrpos — Find position of last occurrence of a char in a string
- strspn — Finds the length of the first segment of a string consisting entirely of characters contained within a given mask.
- strstr — Find first occurrence of a string
- strtok — Tokenize string
- strtolower — Make a string lowercase
- strtoupper — Make a string uppercase
- strtr — Translate certain characters
- substr_compare — Binary safe comparison of 2 strings from an offset, up to length characters
- substr_count — Count the number of substring occurrences
- substr_replace — Replace text within a portion of a string
- substr — Return part of a string
- trim — Strip whitespace (or other characters) from the beginning and end of a stringstrncmp — Binary safe string comparison of the first n characters * ucfirst — Make a string's first character uppercase
- ucwords — Uppercase the first character of each word in a string
- vfprintf — Write a formatted string to a stream
- vprintf — Output a formatted string
- vsprintf — Return a formatted string
- wordwrap — Wraps a string to a given number of characters

Strings in C & C++

- Strings are not primitive: arrays of char
- No simple variable assignment

```
char line[MAXLINE];
char *p, q;
p = &line[0];
```

- Have to use a library routine, strcpy()

```
if(argc==2) strcpy(filename, argv[1]);
```

- **`strcpy()`** no bounds checking ⟹ possible overflow attack
- C++ provides a more sophisticated string class

Strings in other languages

- SNOBOL4 is a string manipulation language
  - Strings: primitive data type
  - Includes many basic operations
  - Includes built-in pattern-matching operations
- Fortran and Python
  - Primitive type with assignment and several operations

Strings in other languages

- Java: Primitive via the String class
- Perl, JavaScript, Ruby, and PHP
  - Provide built-in pattern matching, using regular expressions
  - Extensive libraries
- Lisp:
  - A type of *sequence*
  - Unlimited length, mutable

String implementation

- Strings seldom supported directly by hardware
- Software ⇒ implement strings
- Choices for length:
  - **Static**: set at creation time, then unchanged (FORTRAN, COBOL, Java's/.NET's String class)
  - **Limited dynamic**: max length set at creation, actual length varies up to that (C, C++)
  - **Dynamic**: no maximum, varies at runtime (SNOBOL4, Perl, JavaScript, Lisp)
- Some languages provide all three types - Ada, DBMS (`Char, Varchar(n), Text/Blob`)

String implementation

- Static length: compile-time descriptor
- Limited dynamic length:
  - may need a run-time descriptor
  - C/C++: null (0) terminates string
- Dynamic length:
  - need run-time descriptor
  - computationally inefficient - allocation/de-allocation problem

Compile- and run-time descriptors

Static string
Length
Address

Compile-time descriptor for static strings

Limited dynamic string
Maximum length
Current length
Address

Run-time descriptor for limited dynamic strings

What about dynamic strings?

Immutable strings

- Many languages allow strings to be changed
  - Character replacement
  - Insertion of slices
  - Changes of length
  - C, Lisp, many others
- Others have **immutable** strings
  - Cannot change them
  - To make a "change", have to create new string
  - Python, Java, .NET languages, C++ (except C-like strings)

Immutable strings

- Advantages of immutable strings:
  - "Copying" is fast — just copy pointer/reference
  - Sharing of strings is safe — even across processes
  - No inadvertent changes (via, e.g., aliases or pointers)
- Disadvantages:
  - For minor changes, still have to copy the entire string
  - Memory management (manual or GC)

User-Defined Ordinal Types

User-defined ordinal types

- **Ordinal type:** range of possible values mapped to set of (usually positive) integers
- Primitive ordinal types - e.g., integer, char, Boolean...
- User-defined ordinal types:
  - **Enumerations**
  - **Subranges**

Enumerations

- Define all possible values in definition
- Values are essentially named constants
- C#:

```
enum days {mon, tue, wed, thu, fri, sat, sun};
```

- Pascal example (with subranges)

```
Type
  Days = (monday,tuesday,wednesday,thursday,

             friday, saturday,sunday);
  WorkDays = monday .. friday;
  WeekEnd = Saturday .. Sunday;
```

Enumerations

- First appeared in Pascal and C
- Pascal-like languages: can subscript arrays using enumerations

```
var schedule : array[Monday..Saturday] of string;
var beerPrice : array[Budweiser..Guinness] of real;
```

- Primary purpose of enumerations: enhance readability
- Some languages treat enums as integers and perform implicit conversions
- Others (e.g., Java, Ada): strict type-checking, require explicit conversions (**casting**)

Enumerations

- Languages not supporting enumerations:
  - Major scripting languages - Perl, JavaScript, PHP, Python, Ruby, Lua
  - Java, for first 10 years (until version 5.0)
- Design issues
  - Can an enumeration value appear in more than one type?
  - If so, how is this handled?
  - Are enumeration values coerced to integers?

```
for (day = Sunday; day <= Saturday; day++)
```

  - Any other type coerced to an enumeration type?

```
day = monday * 2;
```

Why use enumerated types?

- Readability - e.g., no need to code a color as a number
- Reliability - compiler can check:
  - operations (don’t allow colors to be added)
  - range checking
  - Some languages better than others at this
  - E.g., Java, Ada, C# - can't coerce to integers
  - Ada, C#, and Java 5.0 provide better support

Subranges

- Subrange: ordered, **contiguous** subsequence of an ordinal type
- E.g., 12 ..18 — subrange of integer type
- E.g. - Ada:

```
type Days is (mon, tue, wed, thu, fri, sat, sun);
subtype Weekdays is Days range mon..fri;
subtype Index is Integer range 1..100;

Day1: Days;
Day2: Weekday;
Day2 := wed;
Day1 := Day2;
```

Why use subranges?

- Readability - way to explicitly state that variable can only store one of a range of values
- Reliability - compile-time, run-time type checking

User-defined ordinal types:

- Enumeration types: usually implemented as integers
- Issue: how well does the compiler hide implementation?
- Subrange types: implemented like parent types
- Run-time checking via code inserted by the compiler

Arrays

Array Type

- Array:
  - collection of homogeneous data elements
  - each element: identified by position relative to the first element
- Except for strings, arrays are the most widely-use non-scalar data type

Array Design Issues

- What types are legal for subscripts?
- Are subscripting expressions in element references range checked?
- When are subscript ranges bound?
- When does allocation take place?
- What is the maximum number of subscripts?
- Can array objects be initialized?
- Are any kind of **slices** supported?

Array Indexing

- **Indexing** (*subscripting*): mapping from indices to elements

  `array_name (index_value_list)` → an element

- Index syntax
  - FORTRAN, PL/I, Ada, Basic, Pascal: **`foo(3)`**
  - Ada: uses **`bar(4)`**
    - to explicitly show uniformity between array references and function calls
    - why? both are mappings
  - Most other languages use brackets
  - Some are odd: e.g., Lisp:

```
(aref baz 7)
```

Array index type

- FORTRAN, C: integer only
- Ada, Pascal : any ordinal type, e.g., integer, integer subranges, enumerations, Boolean and characters
- Java: integer types only

Array index range checking

- Tradeoff between safety, efficiency
- No bounds checking ⇒ buffer overflow attacks
- C, C++, Perl, and Fortran — no range checking
- Java, ML, C# specify range checking
- Ada: default is range checking, but can be turned off

Arrays in Perl

- Array names in Perl start with @
- Elements, however, are scalars ⟹ array element references start with $
- Negative indices: from end

```
@friends = ("Rachel", "Monica", "Phoebe",
            "Chandler", "Joey", “Ross");
# prints "Phoebe"
print $friends[2];
# prints "Joey"
print $friends[-2];
```

Lower bounds

- Some are **implicit**
  - C-like languages: lower bound is always 0
  - Fortran: implicit lower bound is 1
- Other languages allow user-specified lower bounds
  - Pascal-like languages, some Basic variants: arbitrary lower bounds
  - Some Basic variants: **Option Base** statement sets implicit lower bound

Subscript binding and array

- **Static**:
  - subscript ranges statically bound
  - storage allocation static (compile time)
  - efficient with respect to time — no dynamic allocation
- **Fixed stack-dynamic**:
  - subscript ranges: statically bound
  - allocation at runtime function invocation
  - efficient with respect to space (but slower)

Subscript Binding and Array

- **Stack-dynamic**:
  - subscript ranges are dynamically bound
  - storage allocation is dynamic (at run-time)
  - flexible — array size isn't needed to be known until array is used
- **Fixed heap-dynamic**:
  - similar to fixed stack-dynamic
  - storage binding is dynamic — but fixed after allocation
  - i.e., binding done when requested, storage from heap

Subscript Binding and Array

- **Heap-dynamic**:
  - binding of subscript ranges, storage allocation is dynamic
  - can change any number of times
  - flexible —arrays can grow or shrink during program execution

Sparse Arrays

- **Sparse array**: some elements are missing values
- Some languages support sparse arrays: JavaScript, e.g.
  - subscripts needn't be contiguous
  - e.g.,

```
var myColors = new Array ("Red, “Green",
                          "Blue", "Indigo",
                          "Violet");
myColors[15] = “Orange“;
```

Subscript binding and array

- C and C++
  - Declare array outside function body or using `static` modifier ⟹ static array
  - Arrays declared in function bodies: fixed stack-dynamic
  - Can allocate fixed heap-dynamic arrays
- C# — `ArrayList` class provides heap-dynamic
- Perl, JavaScript, PHP, Python, and Ruby: heap-dynamic
- Lisp: fixed heap-dynamic or heap-dynamic (although adjusting size requires function call)

Array initialization

- C, C++, Java, C#

```
int list [] = {4, 5, 7, 83}
```

- Character strings in C and C++

```
char name [] = “freddie";
char name [] = {‘f’, ‘r’, ‘e’, ‘d’, ‘d’, ‘i’, ‘e’};
```

- Arrays of strings in C and C++

```
char *names [] = {"Bob", "Jake", “Joe”};
```

- Java initialization of **String** objects

```
String[] names = {"Bob", "Jake", "Joe"};
```

Array initialization

- Ada

```
Primary : array(Red .. Violet) of Boolean =
              (True, False, False, True, False);
```

Heterogeneous arrays

- **Heterogeneous array:** elements need not be the same type
- Supported by Perl, Python, JavaScript, Ruby, PHP, Lisp
- PHP:

```
$fruits = array ("fruits"  => array("a" => "orange",
                                    "b" => "banana",
                                    "c" => "apple"),
                 "numbers" => array(1, 2, 3, 4, 5, 6),
                 "holes"   => array("first",
                                  5 => “second",
                                  "third"));
```

Initialization with *comprehensions*

- **Intensional** rather than **extensional** definition of list
- First appeared in Haskell, now in Python
- Function is applied to each element of an array or thing in iterator to construct a new array:

```
list = [x ** 2 for x in range(12) if x % 3 == 0]
```

⟹ puts [0, 9, 36, 81] in list

- Smalltalk: block of code could be passed to any iterator
- Lisp/Scheme: *mapping* functions do similar thing:

```
(remove-if 'null (mapcar '(lambda (a)
                            (if (= 0 (mod a 3))
                             (expt a 2)))
                             '(0 1 2 3 4 5 6 7 8 9 10 11)))
```

Automatic array initialization

- Some languages — pre-initialize arrays
  - E.g., Java, most BASICs
  - Numeric values set to 0
  - Characters to \0 or \u0000
  - Booleans to false
  - Objects to null pointers
- Relying on automatic initialization: dangerous programming practice

Array operations

- Array operations work on the array as a single object
  - Assignment
  - Concatenation
  - Equality / Inequality
  - Array slicing

Array operations

- C/C++/C# : none
- Java: assignment
- Ada: assignment, concatenation
- Python: numerous operations, but assignment is reference only
- Deep vs shallow copy
  - **Deep copy:** a separate copy where all elements are copied as well
  - **Shallow copy:** copy reference only

Array operations – implied

- Fortran 95 — “elemental” array operations
  - Operations on the elements of the arrays
  - Ex: `C = A + B` $\Longrightarrow$ `C[i] = A[i] + B[i]`
  - Provides assignment, arithmetic, relational and logical operators
- APL has the most powerful array processing facilities of any language
  - operations for vectors and matrixes
  - unary operators (e.g., to reverse column elements, transpose matrices, etc.)

Jagged arrays

- Most arrays: *rectangular*
  - multidimensional array
  - all rows have same number of elements (equivalently, all columns have the same number of elements)
- **Jagged arrays:**
  - rows have varying number of elements
  - possible in languages where multidimensional arrays are really arrays of arrays
- C, C++, Java, C#: both rectangular and jagged arrays
- Subscripting expressions vary:

```
arr[3][7]   arr[3,7]
```

Jagged arrays — C#

```
int[][] jaggedArray = new int[3][];
jaggedArray[0] = new int[5];
jaggedArray[1] = new int[4];
jaggedArray[2] = new int[2];
```

- Or

```
int[][] jaggedArray2 = new int[][] {
    new int[] {1,3,5,7,9},
    new int[] {0,2,4,6},
    new int[] {11,22}
};
```

Type signatures

- A **type signature** — usually used to denote the types of a functions' parameters and output
  - E.g., `int foo(int a, float b) {…}`

    has the signature `(int) (int, float)`
- Can also think of type signature applying to data, variables

  E.g., `float x[3][5]`
  - Type of `x: float[][]`
  - Type of `x[1]: float[]`
  - Type of `x[1][2]: float`
  -

Arrays in dynamically typed languages

- Most languages with **dynamic typing:** arrays elements can be of different types
- Implemented as array of pointers
- Many such languages: dynamic array sizing
- Many have built-in support for **lists**
  - one-dimensional arrays
  - not (quite) same as Lisp's lists
- Some languages: **recursive arrays** — array can have itself as an element
- E.g., from Lisp:

```
(setf a '(1 2 3))
(setf (cdr (last a)) a)
a → #1=(1 2 3 . #1#) → (1 2 3 1 2 3 1 2 3 …)
```

Slices

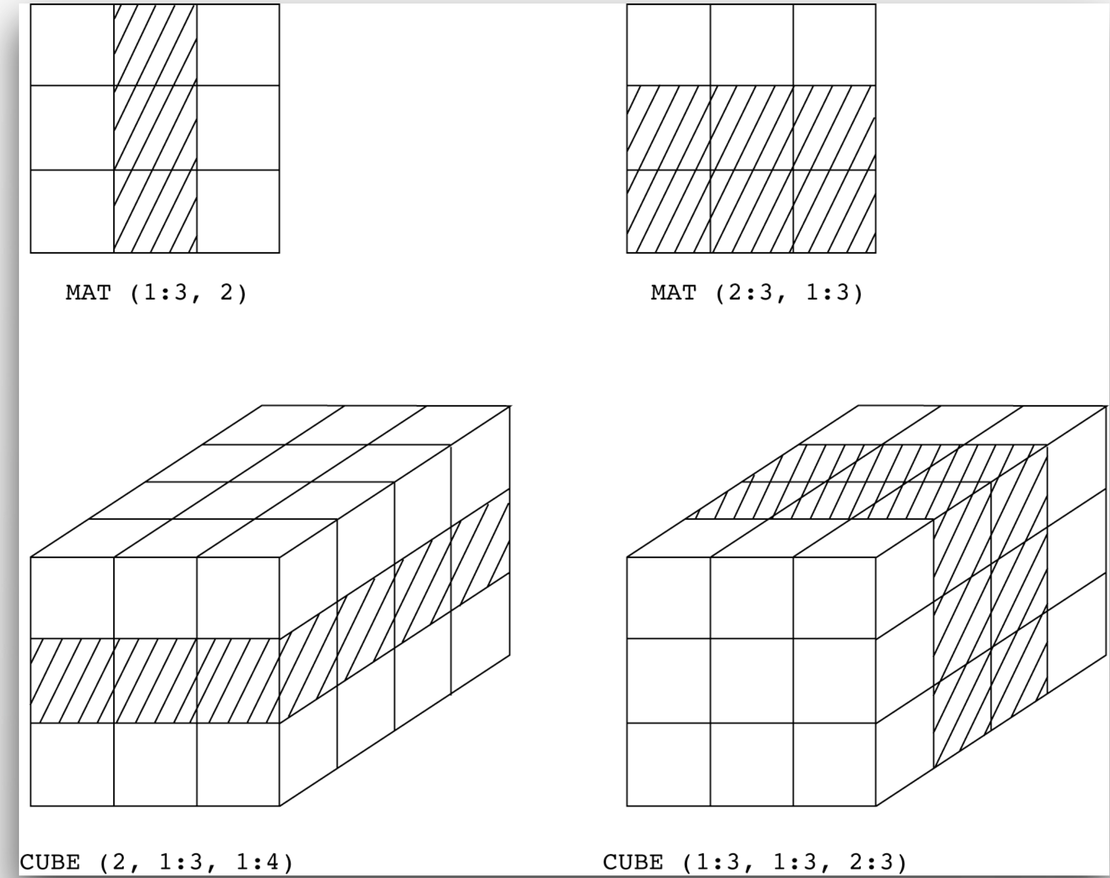
- A **slice** is a substructure of an array
- Just a referencing mechanism

Quick quiz!

1. What are the most common hardware-supported numeric types?
2. What is the primary advantage of using the internal machine representation of integers for arithmetic?
3. What is a significant disadvantage?
4. Why are Booleans rarely represented as single bits even though this is the most space-efficient representation?

Slice Examples

- Fortran 95
  - E.g., Vector(3:6) → four-element array
  - Also allows **strides**: Vector(3:100:2) → slice composed of Vector(3), Vector(5),…, Vector(99)



Slice Examples

- Ruby: `slice` method:

`foo.slice(b,l)` → slice starting at b, length
`list.slice(2, 2)` → third and fourth elements

- Perl: slices with ranges, specific subscripts:

@foo[3..7]   @bar[1, 5, 20, 22]

Python lists and slices

- Example from Python:

```
B = [33, 55, 'hello','R2D2']
```

- Elements accessed with subscripts: `B[0] = 33`
- Slice is a contiguous series of entries:

  Ex: `B[1:2]` `B[1:]` `B[:2]` `B[-2:]`

- Strings are character arrays $\Longrightarrow$ slicing very useful for strings

Array implementation

- Requires more compile-time effort than scalars
- Need **access function** to map subscript expression to address
- Function must support as many dimensions as allowed by language

Vectors

- Access function for single-dimensioned arrays:
  - let:
    - b = starting address of array
    - i = index of desired element
    - l = lower bound (0 for C-like languages)
    - s = element size
  - Then address A of desired element:

$$A = b + (i - l)s$$

Vectors

- Operations performed at runtime
- For static arrays, can rearrange:

$$A = b + is - ls = (b - ls) + is$$

- $(b - ls)$ can be done at compile time → A'
- Access function: $A' + is$
- Can use indirect addressing modes of computer

Array storage order

- Order of storing the columns and rows (2D array):
  - **Row-major order**: each row stored contiguously, then the next, etc.
  - **Column-major order**: columns are stored contiguously, then the next, etc.
- Most languages: row-major order
- Exceptions: Fortran, Matlab

Array addresses

- Given:

```
int A[20][30]
```

an int is 4 bytes, and A[0][0]'s address is 10096,

- what is the address of A[10][12]?

Array addresses

- Given:

```
int A[20][30]
```

an int is 4 bytes, and A[0][0]'s address is 10096,

- what is the address of A[10][12]?

$$
\begin{aligned}
A[10][0] &= b + (i - l)s \\
&= 10,096 + (10 - 0) \times (4 \times 30) \\
&= 10,096 + 10(120) = 11,296 \\
A[10][12] &= 11,296 + (12 - 0) \times 4 \\
&= 11,296 + 48 = 11,344
\end{aligned}
$$

Array storage order

- For higher dimensions: store indices first → last
- E.g., 3D matrix A:
  - store A[0], then A[0]…
  - within A[1]: store A[1,0], then A[1,1], …
  - within A[1,1]: store A[1,1,0], A[1,1,1],…

Array storage order

- Why does this matter?
  - Inefficient to access elements in wrong order
  - E.g., initialize A[128,128] array of 4-byte ints, 4 KB pages using nested loops:

```
for(i=0;i<128;i++)
 for(j=0;j<128;j++)
    A[i,j] = 0;
```

  - Row-major order: 8 rows/page, so 16 pages: A[0,0] → A[7,127] on page 1, A[8,0] → A[15,127] on page 2, …

    ⇒ 16 page faults max

Array storage order

- Column-major order: 8 columns/page, 16 pages:

  A[0,0] , A[1,0], A[2,0], … , A[127,7]

  on page 1,

  A[0,8]→ A[127,15]

  on page 2

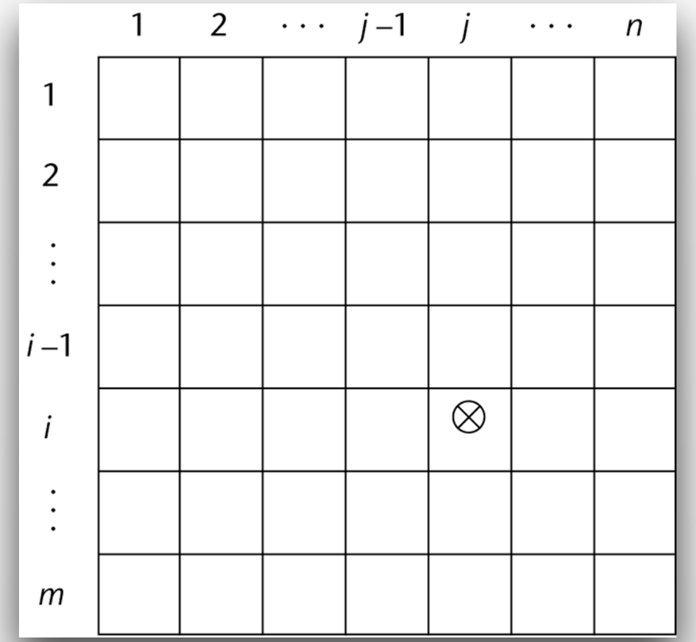
  - Accessing: A[0,0] … A[0,7] on first page, then A[0,8] … A[0,15] on second, etc.
  - 8 page faults max iteration of i ⇒ 8 * 128 = 1024 page faults possible
- Essential to know for mixed-language programming
- Need to know when accessing 2D+ array via pointer arithmetic

Array storage order

- Calculation of element addresses for 2D array A
  - *s:* element size
  - *n:* number of elements/row (= number of columns)
  - *m:* number of elements/column (= number of rows)
  - *b:* base address of A
  - Then:
    - Row-major order:
      - `addr(A[i][j])` $= b + s(ni + j)$
    - Column-major order
      - `address(A[i][j])` $= b + s(mj + i)$

Locating an Element in an n-dimensioned Array

- General format: addr(a[i,j] = b + ((i - lb_r)n + (j - lb_c))s



- For each additional dimension: one more addition and one more multiplication

Compile-time descriptors (**Dope Vectors**)

Array
Element type
Index type
Index lower bound
Index upper bound
Address

Single-dimensioned array

Multidimensioned array
Element type
Index type
Number of dimensions
Index range 1
⋮
Index range *n*
Address

Multi-dimensional array

Associative Arrays

to here, 11/4/14

Associative arrays

- Unordered data elements
- Indexed by **keys**, not numeric indices
- Unlike arrays, keys have to be stored
- Called **associative arrays**, **hashes**, **dictionaries**
- Built-in types in Perl (hashes), Python (dictionaries), PHP, Ruby, Lua (sort of), Lisp (hash tables, association lists)
- Other languages: via classes — .NET's collection class, Smalltalk's dictionaries

Associative arrays: Perl

- **Hashes** — elements are stored in hash tables
- Names begin with %, initialized via an array:

    %hi_temp = (“Monday”, 60, “Tuesday”, 55,…);

  or

    %hi_temp = (“Monday” => 60, “Tuesday” => 55,…);

- Elements accessed via key — elements are scalars, so:

    print $hi_temp{“Tuesday”}; → 55

    $hi_temp{“Wednesday”} = 50;

- Dynamic size

    $hi_temp{“Tuesday”} = 100;

    delete($hi_temp{“Tuesday});

    %hi_temp = {};

Associative arrays: PHP

- **Both** indexed numerically and associative — i.e., ordered collections
- No special naming conventions

  $hi_temps = array("Mon"=>77,"Tue"=>79,“Wed”=>65, …);

  $hi_temps["Wed"] = 83;

  $hi_temps[2] = 83;

- Dynamic size — e.g., add via **$hi_temps[] = 99**
- Rich set of array functions
- Web form processing: query string is in an array ($_GET[]) as are post values ($_POST[])

Associative arrays: Python

- Python: **dictionaries**
- No special naming conventions

```
hi_temps = {'Mon': 77, 'Tue': 79, 'Wed': 65}
hi_temps['Wed'] = 83
```

- Dynamic size: can insert, append, shorten
- Only restriction on keys: immutable

Implementing associative arrays

- Perl
  - **hash** function → fast lookup
  - optimized for fast reorganization
    - 32-bit hash value — but use fewer bits for small arrays
    - need more → add bit (doubling array size), move elements
- PHP
  - hash function
  - stores arrays as linked lists for traversal
  - can have both keys and numeric indices ⟹ can have gaps in numeric sequence
- Python: hash, linked lists as well

Implementing associative arrays

- Lisp
  - **hash tables**
    - built-in data type
    - optimized for size: small table uses list, at some point → true hash table
  - **association lists** (“a-lists”, “assocs”)
    - format: ((key1 . val1) (key2 . val2)…)

```
(setq hi-temp '((monday . 60) (tuesday . 55)…))
```

    - access with assoc:

```
(assoc 'tuesday hi-temp) → (TUESDAY . 55)
        (cdr (assoc 'tuesday hi-temp)) → 55
```

    - implemented as list

Records

Record type

- **Record** composite data type
  - can be heterogeneous
  - identified by name
- Often also called **structs**, **defstructs**, **structures**, etc.
- Record type related to relational/hierarchical databases
- Design issues:
  - How to reference?
  - How to implement (e.g., find element)?

Record type

- First used: COBOL, then PL/I — not in FORTRAN, ALGOL 60
- Common in Pascal-like (“record”) and C-like languages (“struct”)
- Part of all major imperative and OO languages except pre-1990 Fortran
- Similar to classes in OO languages: but no methods
- Not in Java, since classes subsume functionality

Records in COBOL

- **Level numbers** (rather than recursion) to show nested records:

```
01 EMP-REC.
  02 EMP-NAME.
    05 FIRST PIC X(20).
    05 MID   PIC X(10).
    05 LAST  PIC X(20).
  02 HOURLY-RATE PIC 99V99.
```

- Layouts have levels, from level 01 to level 49.
- Level 01 is a special case → reserved for the record level: its name
- Levels from 02 to 49 are all "equal"

Definition of Records: Ada

```
type Emp_Name is record
        First: String (1..20);
        Mid: String (1..10);
        Last: String (1..20);
 end record;

type Emp_Rec is record
        name: Emp_Name;
        Hourly_Rate: Float;
 end record;
```

C example

```
struct employeeType {
    int id;
    char name[25];
    int age;
    float salary;
    char dept;
};
struct employeeType employee;
...
employee.age = 45;
```

- Fields usually allocated in contiguous block of memory
- But actual memory layout is compiler dependent
- Minimum memory allocation not guaranteed

References to record fields

- COBOL

  `field_name OF record_name_1 OF ... OF record_name_n`

  e.g., `FIRST OF EMP-NAME OF EMP-RECORD`

- Other languages: usually "dot notation"

  ```
  recname1.recname2. … .fieldname
  emp_record.emp_name.first;
  ```

- **Fully-qualified references:** include all record names
- COBOL allowed **elliptical reference:** as long as reference is unambiguous:
  - E.g.: SALARY OF EMPLOYEE OF DEPARTMENT
  - could refer to as: SALARY, SALARY OF EMPLOYEE, or fully-qualified

Operations on records

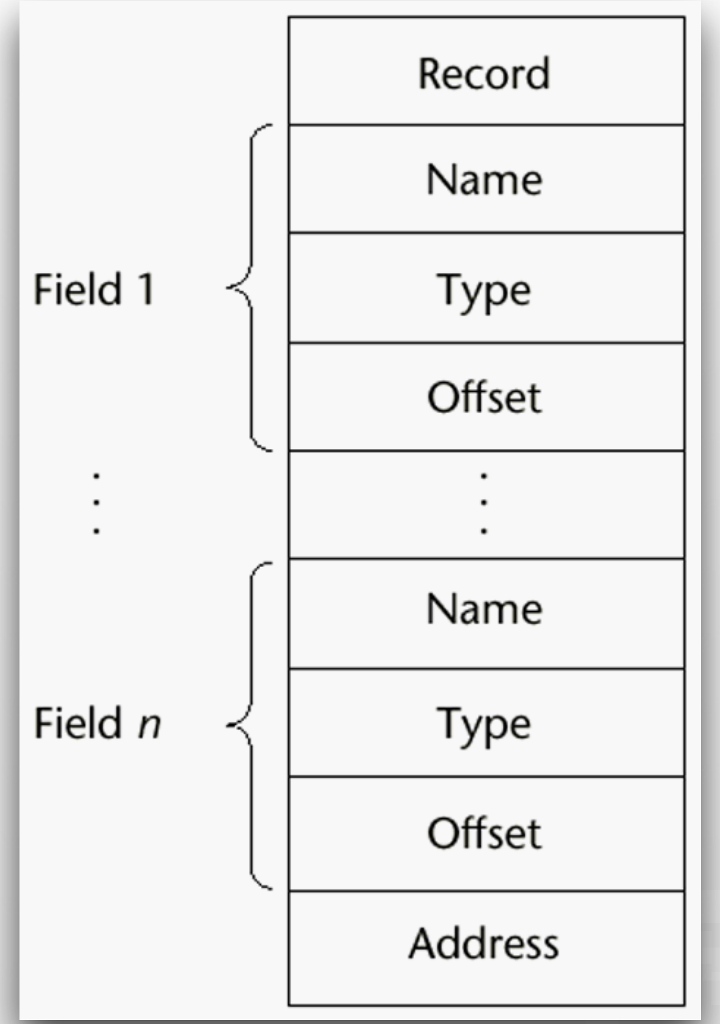
- Assignment : most languages → **memory copy**
- Usually types have to be identical
  - Sometimes can have same structure, even if different names — Ada, e.g.
  - COBOL — **MOVE CORRESPONDING**
    - Moves according to name
    - Structure doesn't have to be same

Operations on records

- Comparison of records:
  - Ada: equality/inequality
  - C, etc.:
    - usually not
    - have to compare field-by-field or…
    - …use memcmp(), etc.

Implementation of Record

- Implemented as contiguous memory
- Descriptors →
  - Compiled languages: need descriptors at compile time only
  - Interpreted: need runtime descriptors



Unions

Unions

- **Union:** data type that can store different types at different times/situations
- E.g.: tree nodes
  - if internal → left/right pointers
  - if leaf → data
- E.g.: vehicle representation
  - if truck, maybe have size of bed, etc.
  - if car, maybe have seating capacity, etc.
- Often in records — subsumed (somewhat) by objects & inheritance
- Design issues
  - Should type checking be required?
  - Should unions be (only) embedded in records?

Unions

- Memory shared between members ⇒ not particularly safe
- C: **free unions**
  - type can be changed on the fly
  - lousy type-checking — even for C:

```
int main() {
    int c;
    union {char a; unsigned char b;} u;
    u.b = 128;
    c = u.b;
    printf("u.b=%d, u.a=%d, c=%d\n", u.b, u.a, c);
}
```

  - called: `u.b=128, u.a=-128, c=128`

Discriminated vs. Free Unions

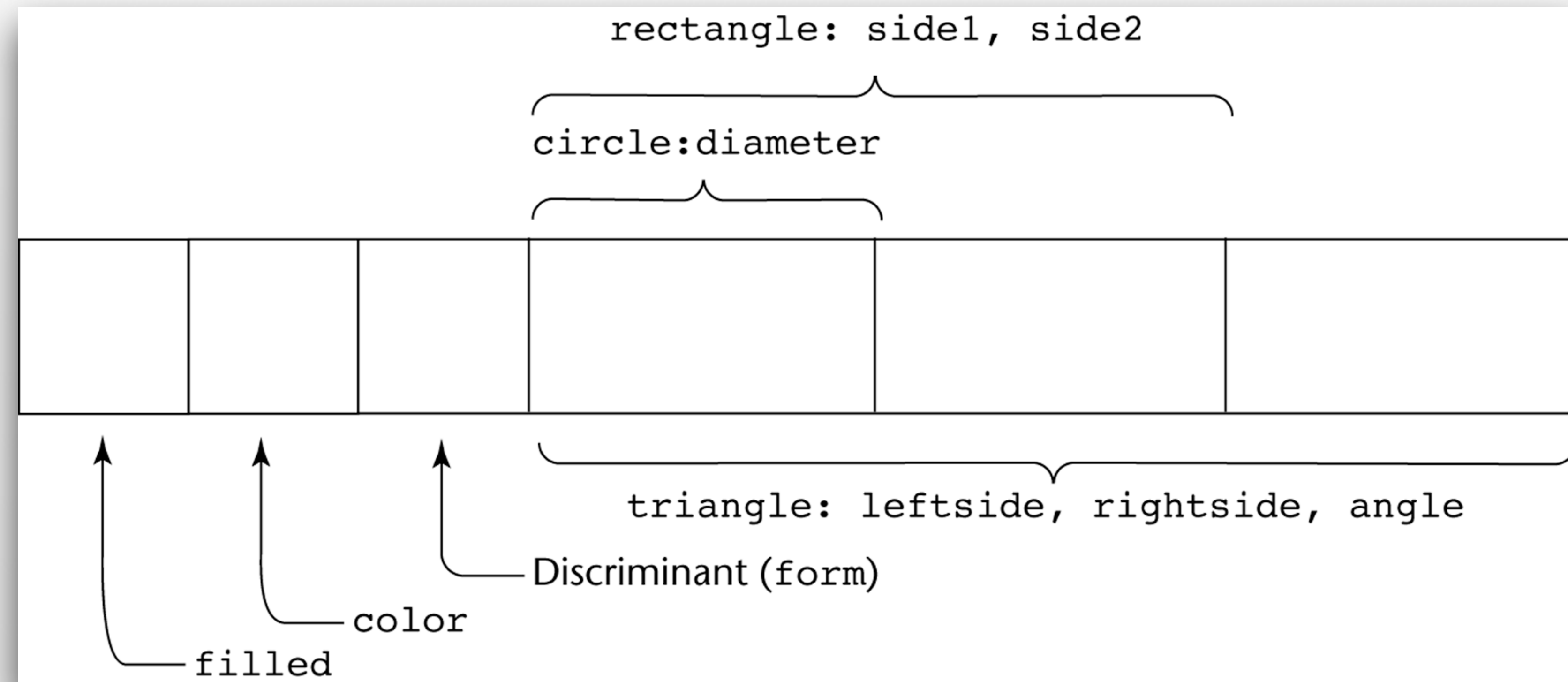
- Free unions: no type checking—FORTRAN, C, C++
- **Discriminated unions:** Pascal, Ada
  - At time of declaration, have to set **discriminant**
  - Type of union is then static → type checking

Ada Unions

```
type Shape is (Circle, Triangle, Rectangle);
type Colors is (Red, Green, Blue);
type Figure (Form: Shape) is record
     Filled: Boolean;
     Color: Colors;
     case Form is
           when Circle => Diameter: Float;
           when Triangle =>
                 Leftside, Rightside: Integer;
                 Angle: Float;
           when Rectangle => Side1, Side2: Integer;
     end case;
end record;
```

Ada Union Type

A discriminated union of three shape variables



Unions

- Free unions are unsafe — major hole in static typing
- Designed when memory was very expensive
- Little or no reason to use these structures today
- Physical memory: much cheaper today
    - Virtual memory → memory space many times the size of actual physical memory
- Java and C# do not support unions
- Ada's discriminated unions are safe — but why use them?
- What to use instead?

Pointers and References

Pointer & reference types

- Pointer holds address or special value (**nil** or **null**)
  - Null → invalid address
  - Usually address 0 $\Longrightarrow$ invalid on most modern hardware
- Two uses:
  - Simulate indirect addressing
  - Provide access to anonymous variables (e.g., from heap)
- **References**:
  - Like pointers — contain memory addresses
  - But operations on them restricted — no pointer arithmetic

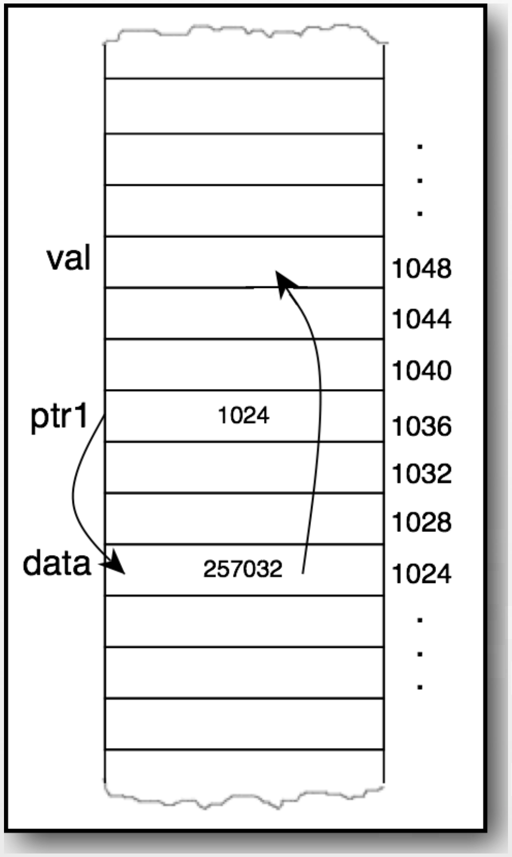
Design issues

- Scope & lifetime?
- Lifetime of heap-dynamic variable pointed to?
- Restricted as to what they point to or not?
- For dynamic storage management, indirection, or both?
- Pointers, reference types, or both?

Pointer operations

- Assignment — pointer's value ← address

```
int data;  int* ptr1, ptr2;
ptr1 = &data;
ptr2 = malloc(sizeof(int));
```



- **Dereferencing:** finding value at location pointed to
  - explicit or implicit (depends on language)
  - C/C++: explicit via *:

```
val = *ptr1;
```

Pointer operations

- Some languages (C, C++): **pointer arithmetic**

```
ptr1 = ptr2++;
```

- Incrementing a pointer: increment depends on type!

```
int a[3];
int* p = &a;   //p → &a[0]
p++            //p → &a[0] + 4 = a[1]
```

Problems with pointers

- Pointers can ⇒ aliases
  - Readability
  - Non-local effects
- **Dangling pointers**
  - Pointer `p` points to heap-dynamic variable
  - Free the variable, but don't zero `p`
  - What does it point to?
- Lost heap-dynamic variables ("**garbage**")
  - Pointer `p` points to heap-dynamic variable
  - Pointer `p` set to zero or another address
  - Lost variable ⇒ **memory leak**

Pointers & arrays: C

- Pass an array variable to function ⟹ behaves like a pointer

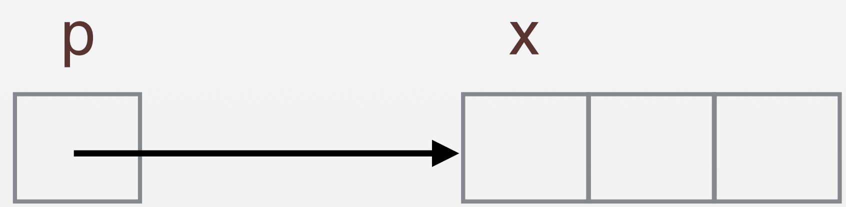
```
float sum(float a[], int n) {
   int i;
   float s = 0.0;
   for (i=0; i<n; i++)
      s += a[i];
   return s;
}
```

```
float sum(float *a, int n) {
   int i;
   float s = 0.0;
   for (i=0; i<n; i++)
      s += *a++;
   return s;
}
```

Pointers & arrays: C

- Common misconception: pointers and arrays are equivalent in C:

```
int x[3] = {1, 2, 3};
int *p = &x[0]; //p points to first element of x
if (p[1] == x[1])
    return 1;
else
    return 0;
```



- Returns 1
- But:
  - x & p have different storage — maybe different scopes, lifetimes
  - p doesn't always have to point to x's storage
  - p can be indexed, but x cannot be assigned a new address

C pointer arithmetic

```
float stuff[100];
float *p;
p = stuff;

*(p+5) ≡ stuff[5]
*(p+i) ≡ stuff[i]
```

C pointer arithmetic

String copy:

```
void strcopy (char *s, char *t) {
    // Kernighan & Ritchie classic:
    while (*s++ = *t++) ;
}
```

Push, pop (where p → next element — initially base of array):

```
*p++ = value;  //push
val = *--p;     //pop
```

Void pointers

- C/C++: pointers of type `void*` allowed
- These are **generic pointers** — can be used to get around type system
- But cannot be explicitly dereferenced

```
void* p;
float a;
float num = 123.456;
p = &num;
a = *(float*)p;
```

- Must **cast** to a `float*` type first, then dereference

Pointer representation

- Prior to ANSI C — pointers and integers were often treated as being the same
- Intel x86 — pointers somewhat more complex: e.g., segment and offset
- Since ANSI C — programmers don't worry too much about the implementation

References

- **References:** similar to pointers … but whereas:

```
int a = 1;
int* p;
printf("size of int = %i\n”,(int)sizeof(int));
p = &a;
printf("p=%lu, *p=%i\n", (unsigned long)p, *p);
```

⇒ call it: size of int = 4

p=140732783793308, *p=1

- …a reference can’t:
  - be printed
  - participate in “reference arithmetic”
  - be dereferenced manually (usually)

References

- C++ includes **reference** — special type of pointer
- Primarily used for formal parameters
- Constant pointer, always **implicitly dereferenced**
- Used to pass parameters by reference (rather than value)

```
void square(int x, int& result) {
    result = x * x;
}

int myint = 12;
int z;
square(myint, &z);
```

⇒ z == 144 afterward

References

- Java — extends C++ references $\Longrightarrow$ replace pointers completely
- References aren't treated as addresses — they just *refer to* objects
- C# — both Java-like references and C++ -like pointers

Reference implementation

- Implementation depends on compiler/interpreter
- Not usually part of specification of language
- E.g., early Java VM:
  - Pointers to pointers ← **handles**
  - Can store constant pointers in table, always point to same pointer
  - *That* pointer can change as GC moves object around
  - Disadvantage: speed (2-level indirection)
- Modern Java VMs: addresses (depends, though)

Miscellaneous Types

Symbols

- Primitive type in Lisp, Scheme
- Access to symbol table itself
- No need to code a symbol as an int or string → use primitive data type

Symbols

```
cl-user> 'a
A
cl-user>  (push 'The (quick brown fox))
(THE QUICK BROWN FOX)
cl-user> (set 'a 23)
23
cl-user> a
23
cl-user> (set 'a 'b)
B
cl-user> a
B
cl-user> (set a 4)
4
cl-user> b
4
```

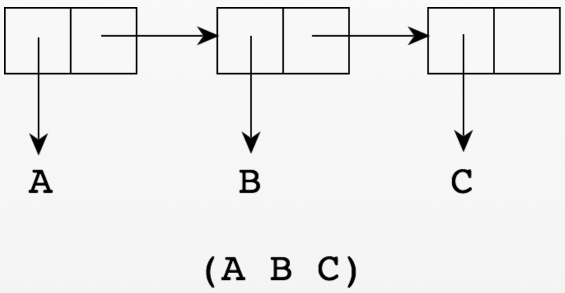
```
CL-USER> (setf exp '(+ (* b b) 10))
(+ (* B B) 10)
CL-USER> (eval exp)
26
```

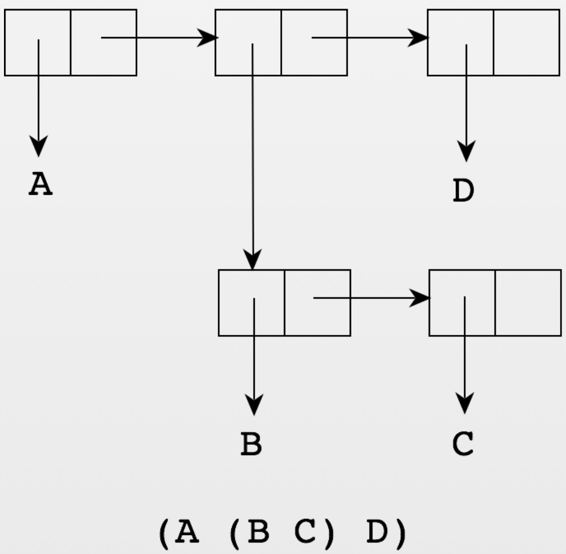
Lists

- Ordered datatypes
- Imply sequential access (but cf. PHP, Python)
- Most: heterogeneous elements
- Nested lists
- Usually implicit linked-lists

Lists: Lisp

- Basic data type in Lisp language family
- Linked list — not indexed
- **Cons cells:** two pointers (references):
    - **car:** points to first element
    - cdr: points to the rest of the list
- Basic element of list (also its own type)
- car, cdr can point to any Lisp object:
    - ⇒ heterogenous lists
    - cdr = null pointer (nil) ⇒ end of list
    - car → cons cell: embedded list
    - either can point to list itself ⇒ circular lists





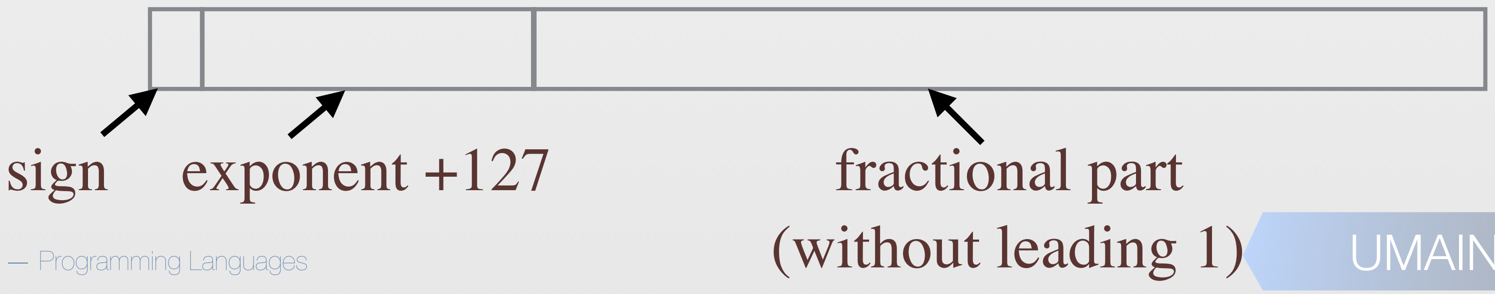
Type Checking

Type checking

- Ensures that operands, operator are compatible
- Operators/operands: also subprograms, assignment
- Compatible types:
  - either explicitly allowed for context
  - can be implicitly converted (**coercion**)
    - following language rules
    - & by code inserted by compiler
- Mismatched types → **type error**

Type conversion

- Can't just treat same bit string differently!
- Ex., 2 stored in variable "foo" in C
  - char foo → 0011 0010 — as ASCII
  - char foo → 0000 0010 — as integer
  - short foo → 0000 0000 0000 0010
  - int foo → 0000 0000 0000 0000 0000 0000 0000 0010
  - float foo → 0100 0000 0000 0000 0000 0000 0000 0000



Type conversions

- **Narrowing conversion:**
  - result has fewer bits
  - ⟹ potential lost info
  - E.g., double → int
- **Widening conversion:**
  - E.g., int → double
  - 32-bit int → 64 bit int — no loss of precision
  - 32-bit int → 32- or 64-bit float — but may lose precision

Type casting & coercion

- **Type cast:** explicit type conversion

```
float z;
int i = 42;
z = (float) i;
```

- **Coercion:** implicit type conversion
  - Rules are language-dependent — can be complex, source of error
  - With signed/unsigned types (e.g., C) — even more complex

C coercion rules

IF	Then Convert
either operand is long double	the other to long double
either operand is double	the other to double
either operand is float	the other to float
either operand is unisgned long int	the other to unsigned long int
the operands are long int and unsigned int and long int can represent unsigned int	the unsigned int to long int
the operands are long int and unsigned int and long int cannot represent unsigned int	both operands to unsigned long int
one operand is long int	the other to long int
one operand is unsigned int	the other to unsigned int

From K&R; also "Unexpected results may occur when an unsigned expression is compared to a signed expression of same size."

Type checking

- Static type bindings → almost all type checking can be static (at compile time)
- Dynamic type binding → runtime type checking
- **Strongly-typed language:**
  - if type errors are almost always detected
  - advantage: type errors caught that otherwise might ⇒ difficult-to-detect runtime errors

Strong/weak typing

- **Weakly-typed:**
  - Fortran 95 — **equivalence** statements map memory to memory, e.g.
  - C/C++: parameter type checking can be avoided, void pointers, unions not type checked, etc.
  - Scripting languages — free use of coercions ⟹ type errors
  - Lisp — though runtime system catches most type errors from coercion, casting, programming errors

Strong/weak typing

- **Strongly-typed:**
  - Ada — unless generic function `Unchecked_Conversion` used
  - Java, C# — but casts, coercions can still introduce errors

Strong typing

- Coercion rules affect strength of typing
- Java has half the assignment coercions of C++
  - no narrowing conversions
  - can still have loss of precision
  - strength of typing still less than (e.g.) Ada

Type Equivalence

Type equivalence

- When are types considered equivalent?
  - Depends on purpose
  - Depends on language
- Pascal report [Jensen & Wirth] on assignment statements:

  "The variable […] and the expression must be of identical type."

  - Problem: didn't say what "identical" meant
  - E.g.: can integer be assigned to an enum var?
  - Standard (ANSI/ISO) fixed this

Type equivalence: C

```
struct complex {
   float re, im;
};
struct polar {
   float x,y;
};
struct {
   float re, im;
} a, b;
struct complex c, d;
struct polar e;
int f[5], g[5]
```

Which are equivalent?

Type equivalence

- Two general types of equivalence:
  - Name equivalence
  - Structural equivalence

Name equivalence

- Two variables are **name equivalent** if:
  - in the same declaration or
  - in declarations using the same type name
- Easy to implement
- Restrictive, though:
  - subranges of integers aren't equivalent to integer types
  - formal parameters have to be same type as actual parameters (arguments)

Structural equivalence

- Two variables are **structurally equivalent** if both types have identical structures
- Flexible
- Harder to implement

Type equivalence

- Some languages are very strict: Ada uses only name equivalence, e.g.
- C — uses both
  - structural equivalence for all types *except* unions and structs where member names are significant
  - name equivalence for unions & structs

Type equivalence: C

```
struct complex {
   float re, im;
};
struct polar {
   float x,y;
};
struct {
   float re, im;
} a, b;
struct complex c, d;
struct polar e;
int f[5], g[5]
```

a, b are (name) equivalent

c,d are name equivalent

e is *not* equivalent to c or d — member names differ

f, g are structurally equivalent

Pointers in C

- All pointers are structurally-equivalent, but
  - object pointed to determines type equivalence
  - e.g., **int * foo; float * baz** — not equivalent
- **void*** pointers…?
- BTW: Array declarations: **int f[5], g[10];** → not equiv.

Ada & Java

- Ada:
  - name equivalence for all types
  - forbids most anonymous types
- Java
  - name equivalence for classes
  - method signatures must match for implementation of interfaces

Functions as Types

Functions as types

- Some languages: can't assign a function to a variable → not **"first-class objects"**
- Why would we want to, though?
    - E.g., graphing routine: pass in function to be graphed
    - E.g., root solver for f(x)
    - E.g., sorting routine, where pass in f(x) to compare items (e.g., generic routine)
    - "Callbacks" in many system APIs

Functions as parameters

- So major need: pass function as a parameter
- Functional language generally have the best support (more later)
- Fortran: function pointers, but no type checking
- Pascal-like languages — function prototype in parameters:

```
Function Newton (A,B : real; function f(x: real): real): real;
```

Function pointers in C

- ANSI C (K&R, 2nd ed.):
  - Functions are not variables
  - Can have pointers to them
  - Can call via pointer
  - Can assign to functions
  - Can return functions

Function pointers in C

- Specification:
  - uses type signatures
  - e.g.:

```
int (*foo)(float, int)
```

```
int cmp_int (int a, b);

int* sort(int array[], int (*cmp) (int, int)
    {… cmp(array[i], array[j]);…}

int temp[20];
…
sort(temp, &cmp_int);
```

- Can be quite messy:

```
int *(*foo) (*int);
```

Java interfaces

- Can do some of same things with **interface**
- **Abstract type** specifying methods class must implement
- Contains method signatures only — no implementations
- Can specify classes that can be passed by specifying the interface

```
public interface RootSolvable {
   double valueAt(double x);
}
public double Newton(double a, double b, RootSolvable f);
```

Functions as first-class objects

- Functions considered **first-class objects** if can be constructed by a function at runtime and returned
- Characteristic of functional languages — not confined to them in modern languages

```
(defun fun-create (op)
    #'(lambda (a b)
        (funcall op a b)))
>> (funcall a 2 3)
5
```

- Even better in Scheme
- Others can do this, too, though: e.g., JavaScript, Python

Functions as first-class objects

- Python example:

```
def make_counter(start=0):
    def counter():
        nonlocal start
        start += 1
        return start
    return counter ← return function
f = make_counter()
f → <function make_counter.<locals>.counter at 0x1022dcd90>
f() → 1
f() → 2
…
```

Heap Management

Start, 11/10/14

Memory & heap

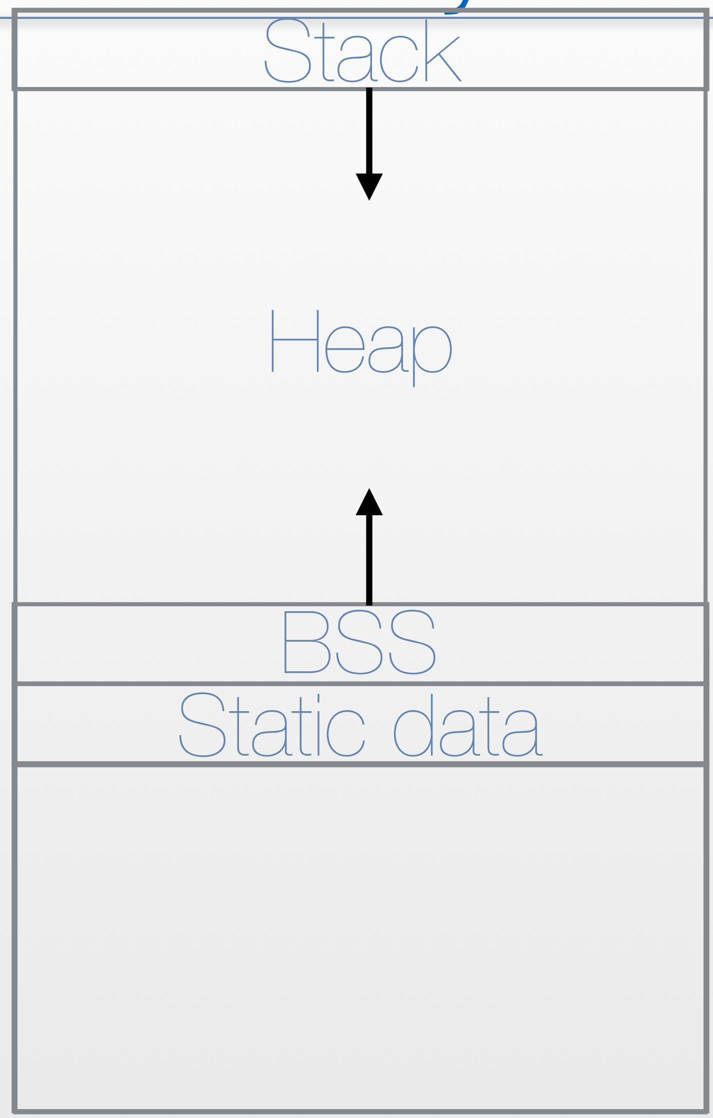
- With respect to memory management and other things:

  > C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off.
  >
  > —Bjarne Stroustrop (creator of C++)

Heap

- Major areas of memory: text, data, stack, heap
- **Text** *(program)* area
- **Data** area
  - Static, initialized variables
  - Dynamic area (**BSS**)
- **Stack** area
- **Heap***:* dynamically-allocated objects

Run-time Memory

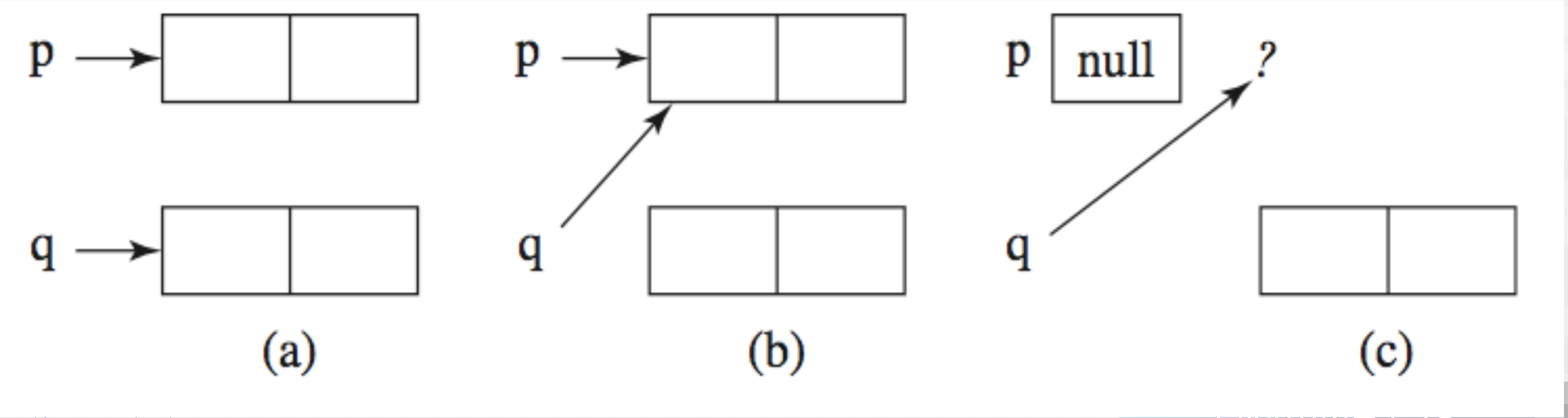


Heap management

- Allocation of data: **`malloc(), new Obj`**
- Deallocation: **`free()`**
- Managing heap:
  - How to find memory for **`malloc()`**?
  - Avoiding dangling pointers
  - Avoiding memory leaks — user or language?
  - If language: how to collect the **garbage**?

Garbage example

```
class node {
  int value;
  node next;
}
node p, q;
p = new node();
q = new node();
q = p;
delete p;
```

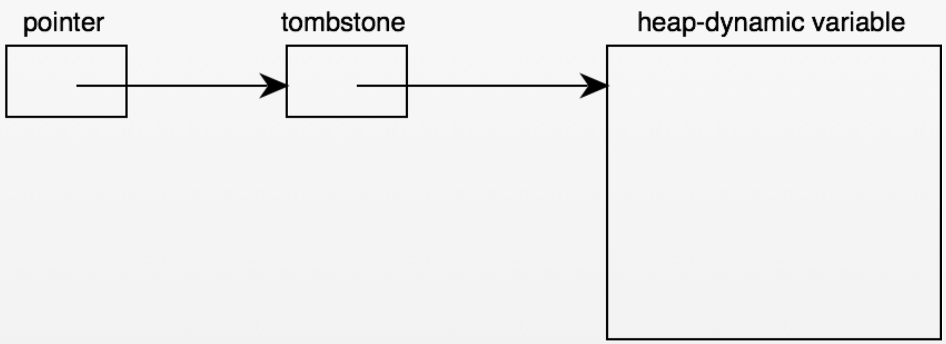


A solution to dangling pointers: Tombstones

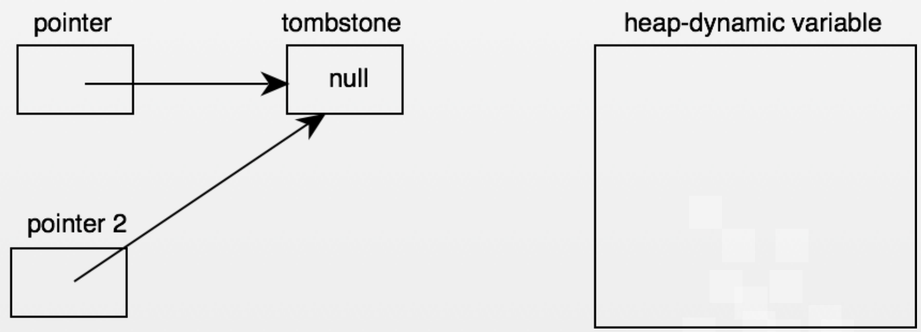
- Allocate a piece of memory from heap → get back a **tombstone**
- Tombstone is a memory cell that itself points to the allocated heap-dynamic variable
- Pointer access is only through tombstones
- When deallocate heap-dynamic variable → tombstone remains, but has null pointer
- Prevents dangling pointers, but…
  - Need extra space for tombstones
  - Every reference to heap-dynamic variable requires one more indirect memory access

A solution to dangling pointers: Tombstones

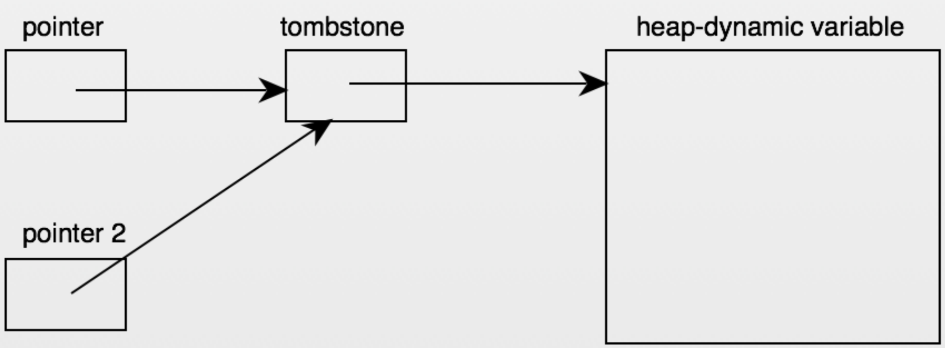
Allocate a heap-dynamic variable:



Deallocate the heap-dynamic variable:



Assign to new pointer:

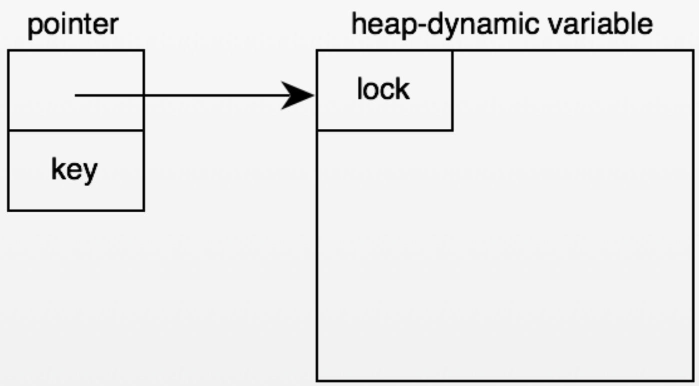


Another solution: Locks and keys

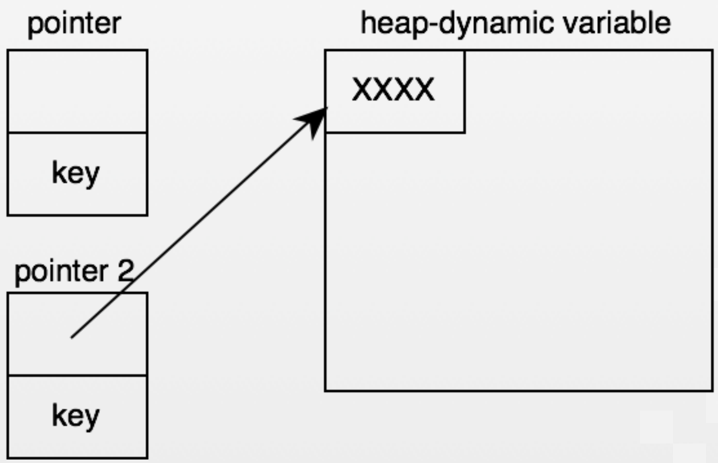
- Heap-dynamic variables = variable + a cell for an integer **lock** value
- Pointers: have both the address and a **key**
- When allocating — create lock, also store in key cell
- Copying pointer: copy key as well
- When accessing: compare lock and key — don't match ⟹ error
- Deallocate heap-dynamic variable: put invalid lock in lock cell
- Future: can't access the data, since lock and key don't match

Another solution: Locks and keys

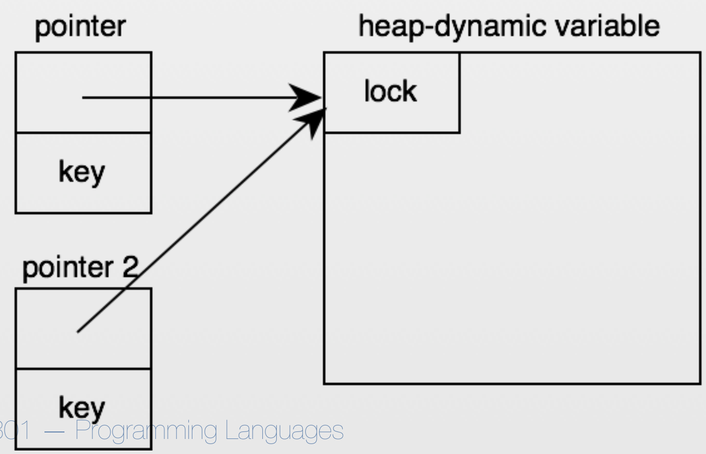
Allocate a heap-dynamic variable:



Deallocate the heap-dynamic variable:



Assign to new pointer (copy key, too):



Garbage collection

- Could be responsibility of programmer
  - E.g., C, C++ (via malloc()), Objective C (on iOS)
  - Pros:
    - Gives programmer complete control of heap
    - Fast: don't have to search for garbage
  - Cons:
    - Makes programming more complex
    - Bugs $\Longrightarrow$ memory leaks — difficult to detect

Garbage collection

- Automatic garbage collection algorithms
  - E.g., Lisp, Java, Python…
  - Pros:
    - No memory leaks
    - Simpler for programmer
  - Cons:
    - Complex
    - Costly with respect to time

GC algorithms

- First designed, used in 1960s: Lisp
- 1990s: OOP, interpreted scripting languages ⟹ renewed interest
- Recall **garbage** = areas of heap no longer in use
- No longer in use = nothing in program points to it
- Functions of GC:
  - Reclaim garbage → **free space list**
  - If non-uniform allocation: **compact** free space as needed to reduce **fragmentation**

GC issues

- How long does it take?
  - Time program is “paused”
  - Full vs incremental
- How much memory does GC itself take?
  - Some schemes may halve the size of available heap

GC issues

- How does it interact with VM?
  - Does GC cause extra page faults?
  - Does GC cause cache misses?
- Can GC be used to improve locality of reference by reorganizing data?
- How much runtime bookkeeping?
  - Does this impact speed?
  - Does this impact available memory?

GC algorithms

- Reference counting
- Mark-and-sweep
- Copy collection

GC: Reference counting

- Occurs when heap block is allocated/deallocated
- Heap is a chain of nodes: **free list**
- Each node has extra field — **reference count**
- Nodes taken from chain, connected to each other via pointers
- When allocated via **`new()`**, object allocated from heap, ref count = 1
- When deallocated via **`delete()`**, ref count decremented
- Reference count = 0 $\Longrightarrow$ return object to heap

GC: Reference counting

- Assignment of pointer variable, say q = p:
  - object pointed to by p → ref count++
  - if q was pointing to object → ref count--
  - if uniform size nodes in linked chain, do this for all linked nodes, too

GC: Reference counting

- Come up with an example in which reference counting would *not* work — i.e., in which garbage would remain.

GC: Reference counting

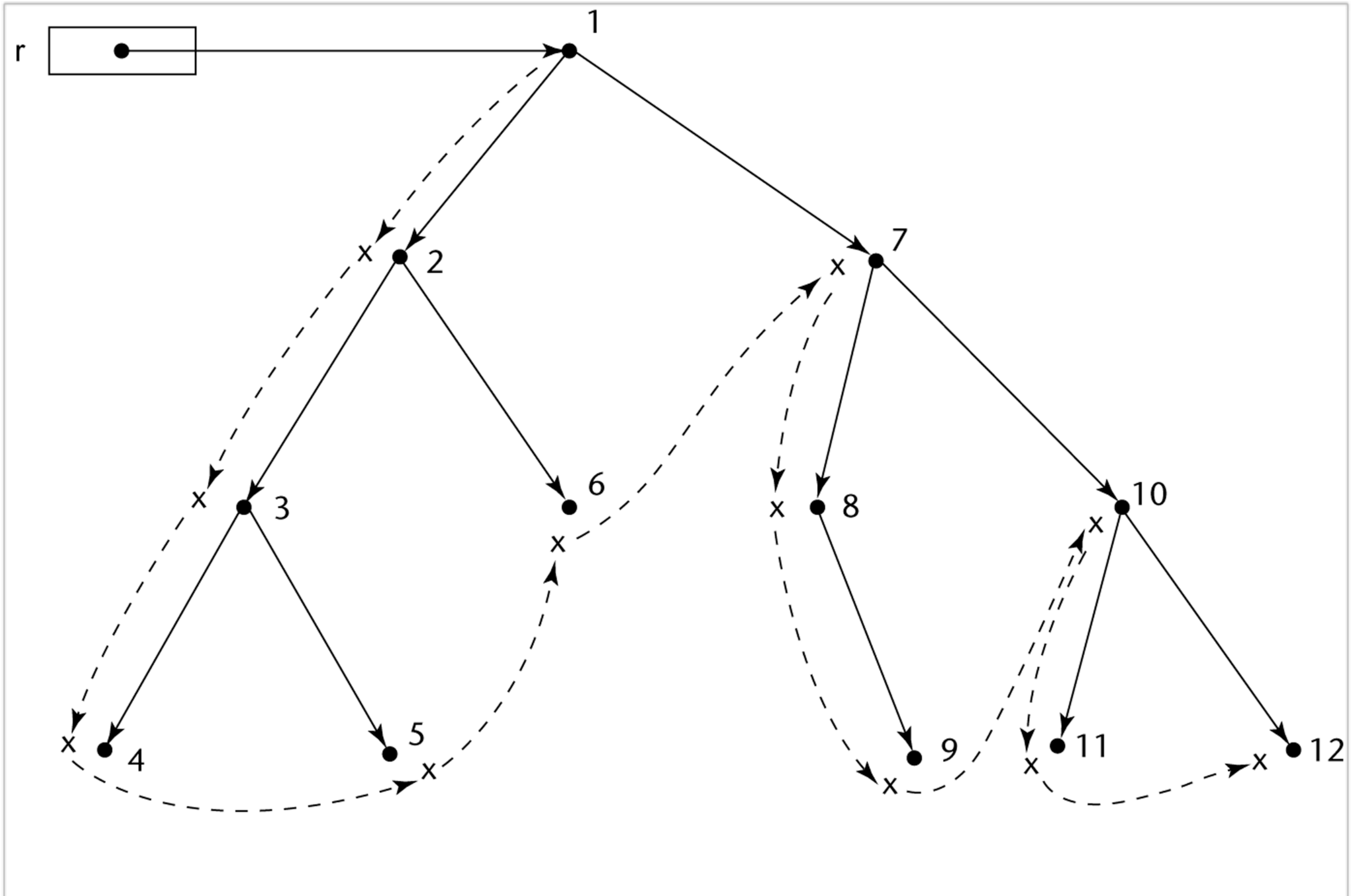
- Pros:
  - Reclaims objects as soon as possible
  - No pauses for GC to inspect heap — intrinsically incremental
- Cons:
  - Requires space for ref counter
  - Increased cost of assignment — bookkeeping
  - Difficulty with circular references

GC: Mark-and-sweep

- Allocate cells from heap as needed
- No explicit deallocation — just change pointer at will
- When heap is full:
  - Find all non-garbage by following (e.g.) all pointers/references in program, marking them as good
  - Return garbage to heap's free list
- Requires two passes over heap
- Also called *tracing collector*

Marking

- Start at every pointer/reference, say *r,* in some known live/root set of pointers:



Dashed lines show the order of node_marking

Sweep

- For every node in the heap:
  - If not marked as in use, then return to free list

Allocation in mark-and-sweep

```
if (free_list == null)
  mark_sweep();
if (free_list != null) {
  q = free_list;
  free_list = free_list.next;
}
else abort('Heap full')
```

Where to start marking?

- *Root set:* set of references that are active
  - Pointers in global memory
  - Pointers on the stack
- May be difficult — e.g., Java has six classes of *reachability* (see, e.g., *here*):
  - strongly reachable
  - weakly reachable
  - softly reachable
  - finalizable
  - phantom reachable
  - unreachable

Problems

- GC can take a *long* time
- Page faults when visiting old (inactive) objects $\Longrightarrow$ more delay
- If non-uniform allocations $\Longrightarrow$ **fragmentation** of heap
- Requires additional space for the mark (not a problem in **tagged architectures**)
- Have to maintain linked list of free blocks

GC: Copy collection

- Trades space for time, compared to mark-and-sweep
- Partition heap into two halves — old space, new space
- Allocate from old space till full
- Then, start from the root set and copy all objects to the new space
- New space now becomes the old space
- No need for reference counts, mark bits
- No need for a free list — just a pointer to end of the allocated area

Copy collection

- Advantages:
  - Faster than mark-and-sweep
  - Heap is always one big block → allocation is cheap, easy
  - Improves locality of reference → objects allocated close to each other, no fragmentation
- Disadvantages:
  - Can only use 1/2 heap space (i.e., more space needed)
  - If most objects are short-lived → good — most won't be copied — but if lots of long-lived objects, spend unnecessary time always copying them back and forth

Generational GC

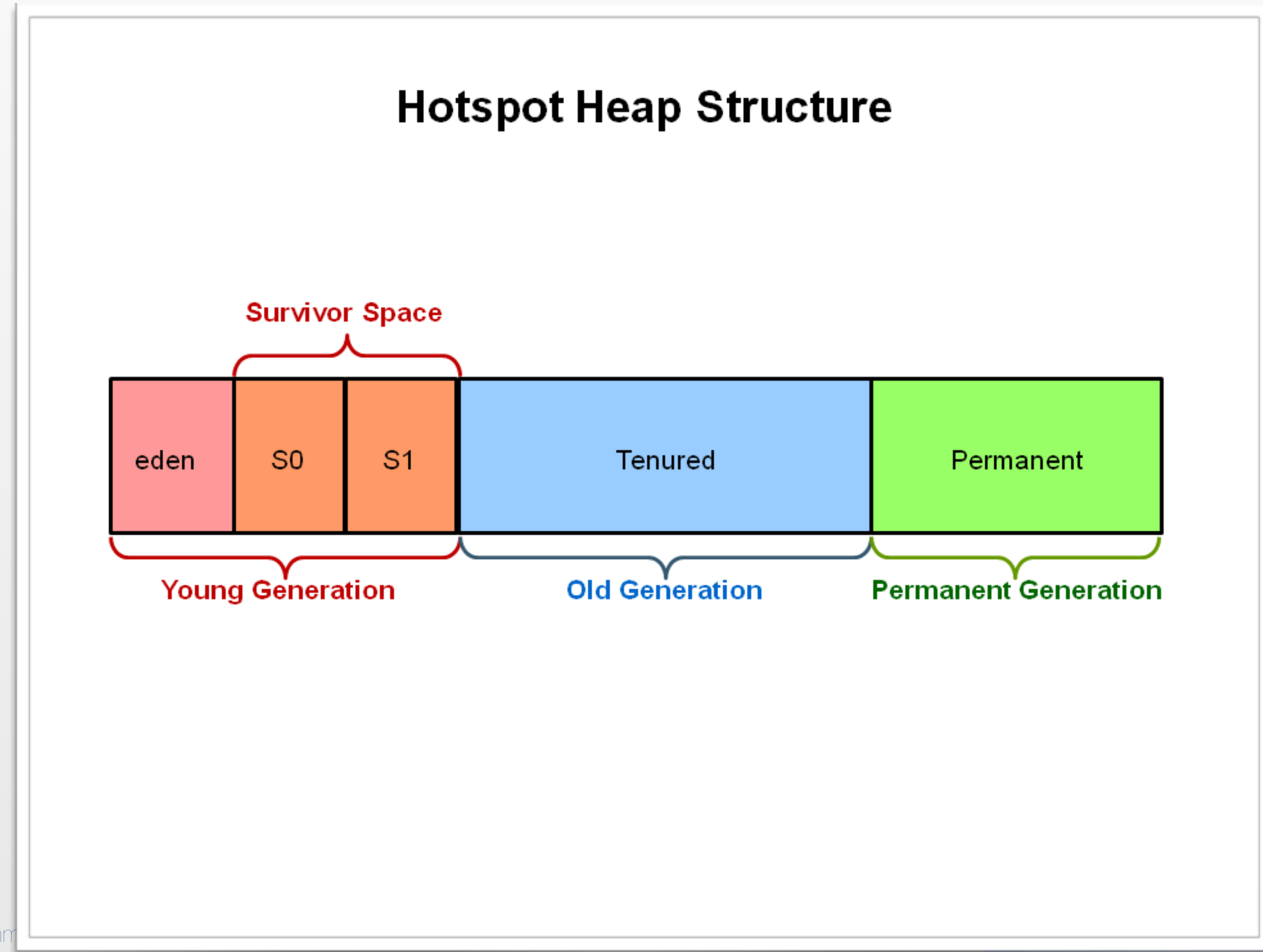
- Empirical studies: most objects in OOP tend to "die young"
- If an object survives one GC, good chance it will become long-lived or permanent
  - Most sources: 90% of GC-collected objects created since last GC
  - Pure copying collector: continues to copy the old objects
- **Generational (ephemeral) GCs***:* make use of this to divide heap into *generations* for different objects

Generational GC

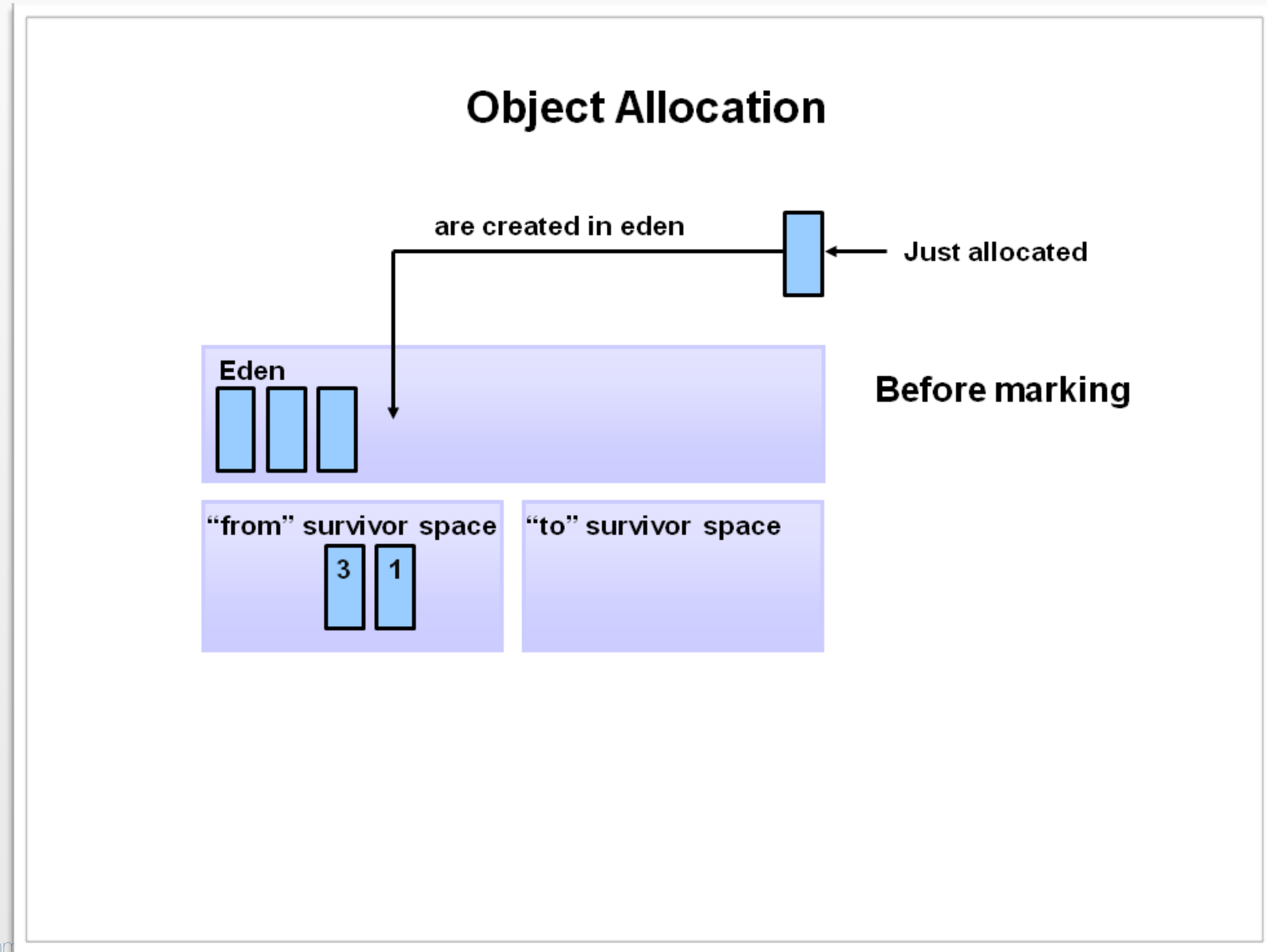
- Heap divided into ***generations***
- Objects start in a generation for new objects
- When object meets some promotion criteria → *promote* to longer-lived generation
- Different algorithms for different generations
- GC:
    - When heap manager needs more space → **minor collection** — only youngest generation considered
    - If this doesn't work → older generations
    - Only fail if all generations have been collected
- Some objects may be unreachable ⟹ need full GC occasionally (mark-and-sweep or copying)

Generational GC: Java

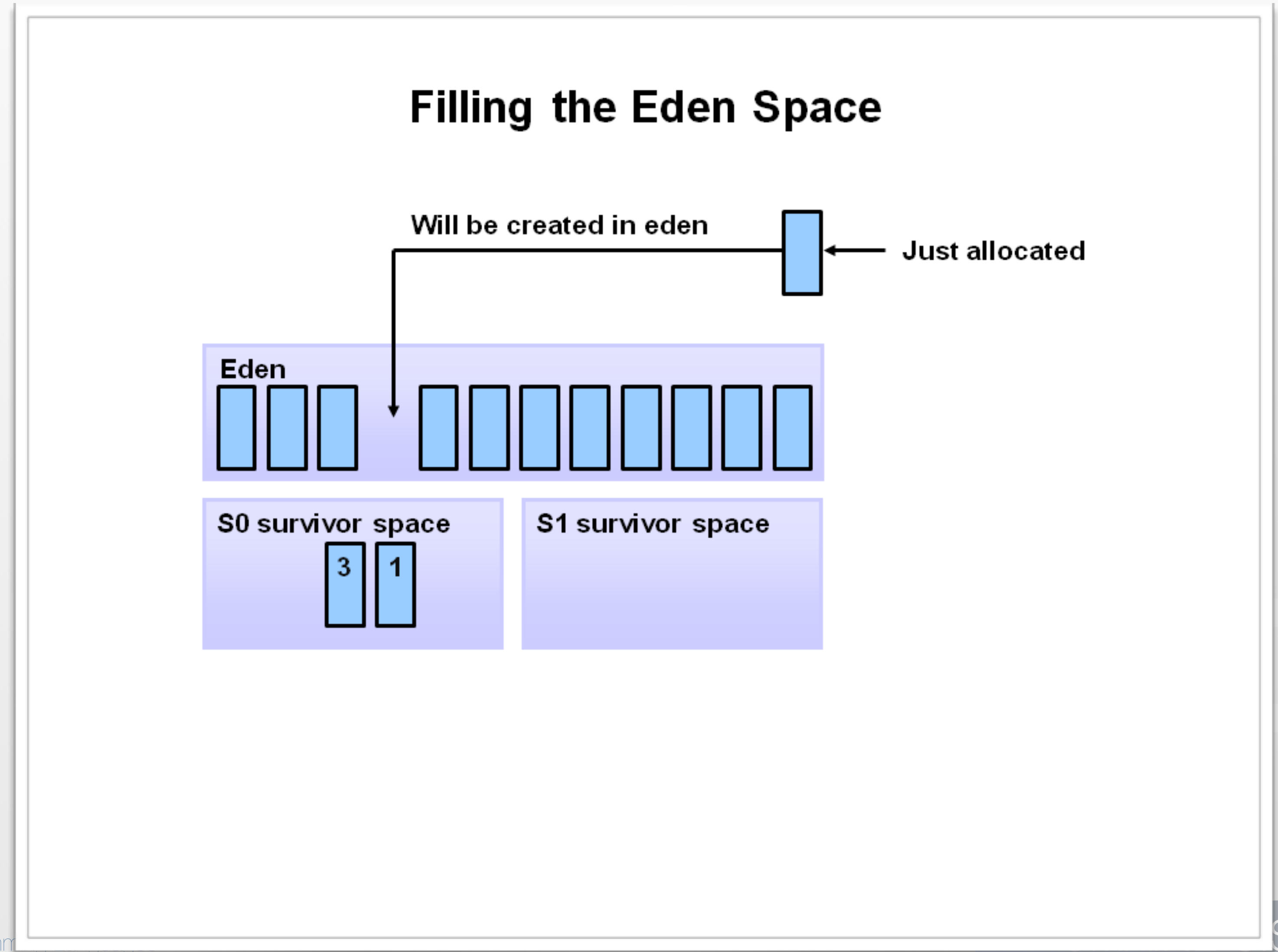
All figures from Oracle: https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html



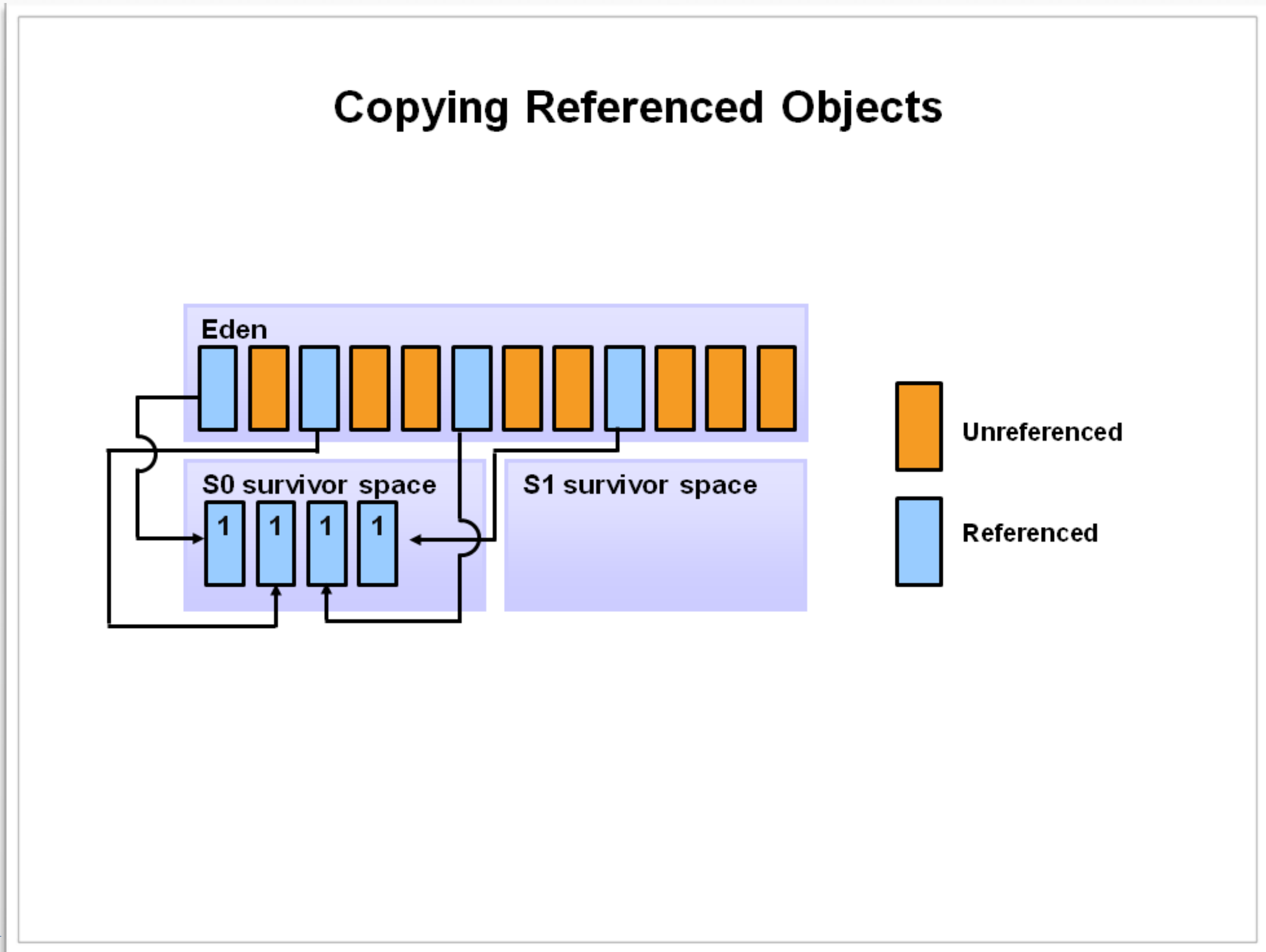
Generational GC: Java



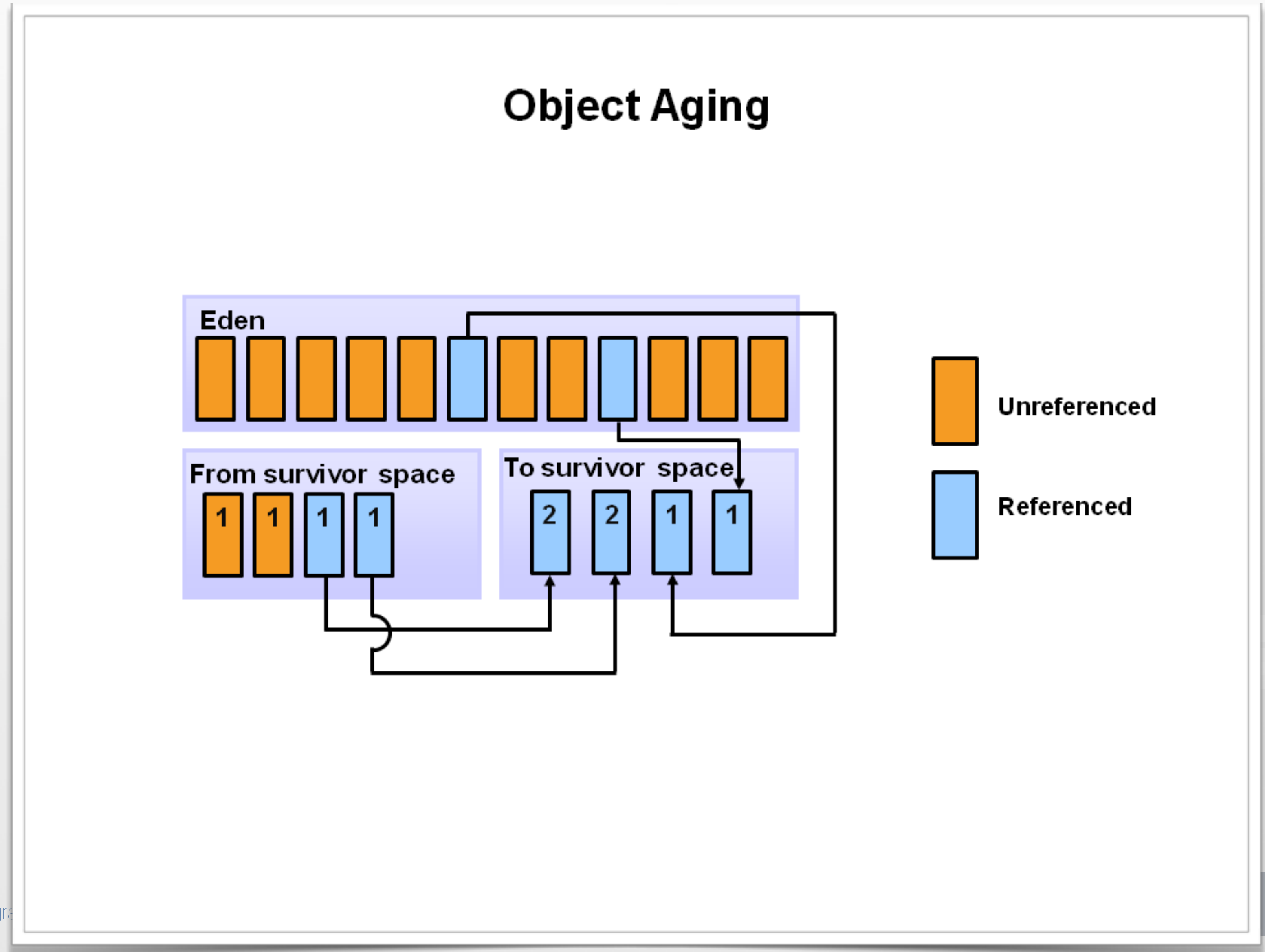
Generational GC: Java



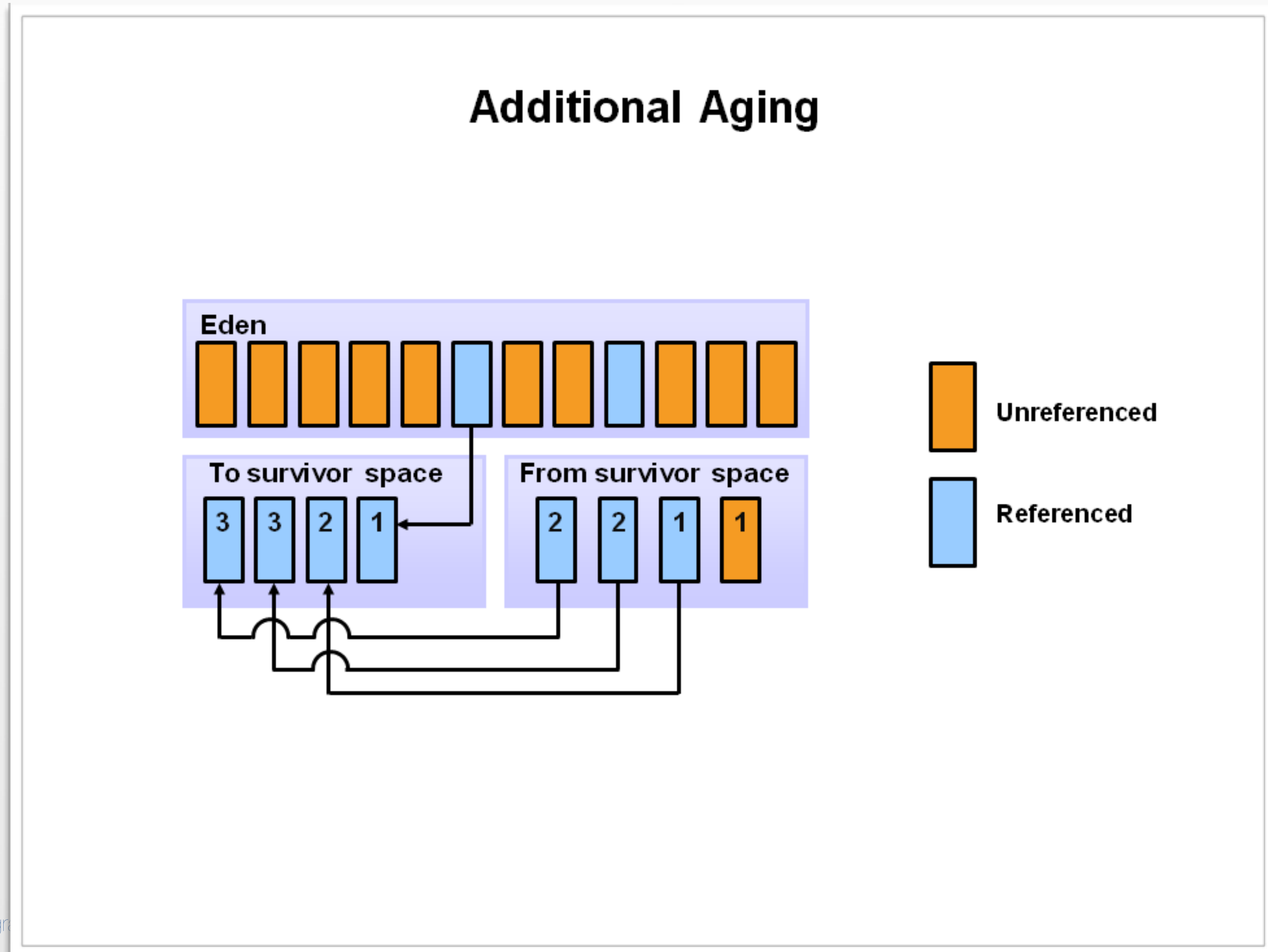
Generational GC: Java



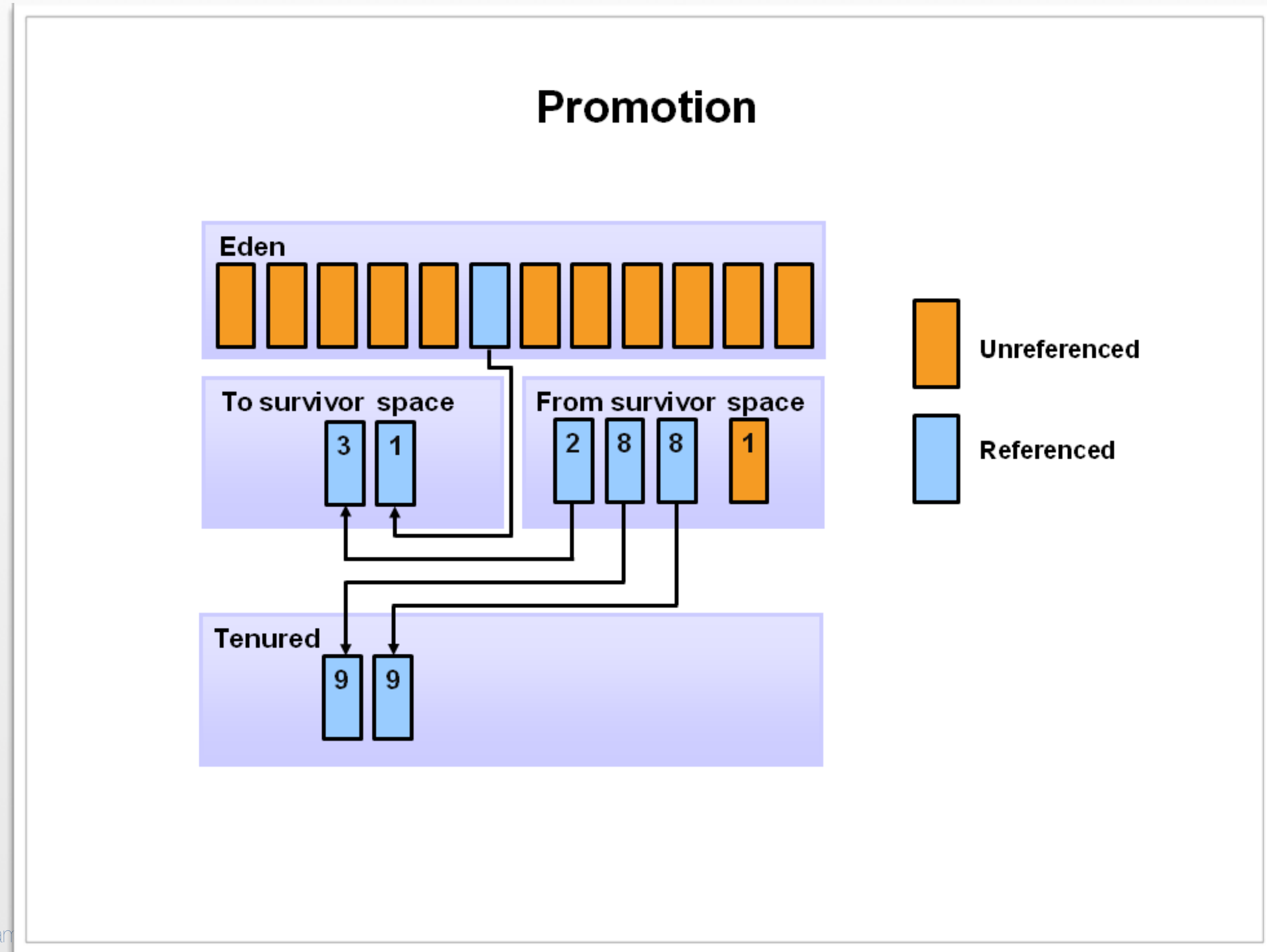
Generational GC: Java



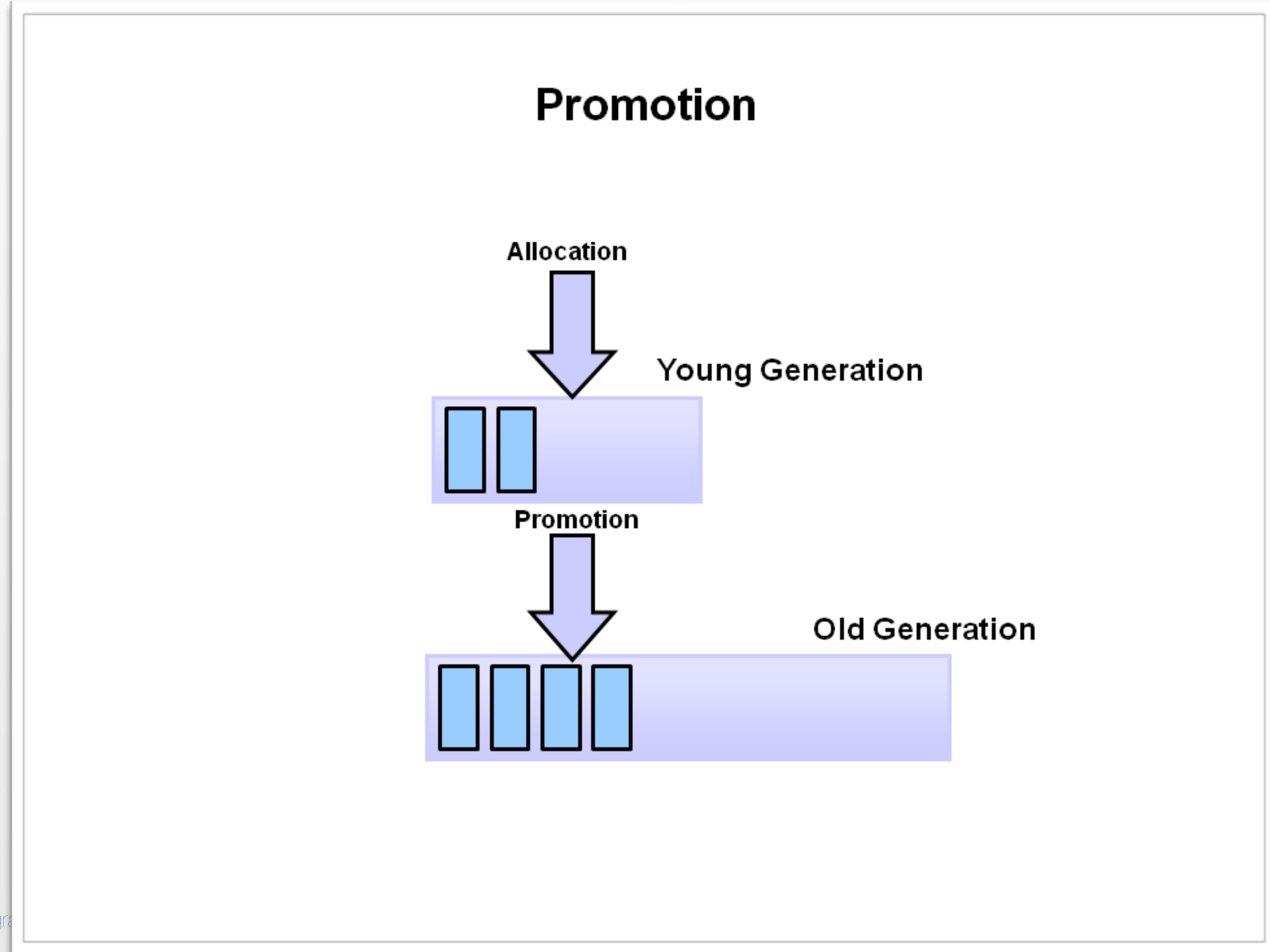
Generational GC: Java



Generational GC: Java



Generational GC: Java



Problem: Intergenerational references

- Generational GC: only visits objects in youngest generation
- But what if object in older generation references object in younger generation that isn't otherwise reachable?
- Solution: explicitly track intergenerational references
  - Easy to do when an object is promoted
  - Harder when change a pointer reference after promotion

Tracking intergenerational references

- Naïve approach: check each pointer assignment for intergenerational reference
- Most common algorithm: **card table** or **card marking**
  - **Card map:** one bit per block of memory (where block usually < VM page)
  - Bit set $\Longrightarrow$ block is **dirty** (written to)
  - When we do a GC, have to consider not just root set, but also any dirty blocks — treat as part of root set
  - If no reference to a younger generation, clear bit