CSPC31: Principles of Programming Languages

Dr. R. Bala Krishnan
Asst. Prof.
Dept. of CSE
NIT, Trichy – 620 015
Ph: 999 470 4853
E-Mail: balakrishnan@nitt.edu

Books

- **Text Books**
  - ✓ **Robert W. Sebesta, *"Concepts of Programming Languages"*, Tenth Edition, Addison Wesley, 2012.**
  - ✓ Michael L. Scott, *"Programming Language Pragmatics"*, Third Edition, Morgan Kaufmann, 2009.
- **Reference Books**
  - ✓ Allen B Tucker, and Robert E Noonan, *"Programming Languages – Principles and Paradigms"*, Second Edition, Tata McGraw Hill, 2007.
  - ✓ R. Kent Dybvig, *"The Scheme Programming Language"*, Fourth Edition, MIT Press, 2009.
  - ✓ Jeffrey D. Ullman, *"Elements of ML Programming"*, Second Edition, Prentice Hall, 1998.
  - ✓ Richard A. O'Keefe, *"The Craft of Prolog"*, MIT Press, 2009.
  - ✓ W. F. Clocksin, C. S. Mellish, *"Programming in Prolog: Using the ISO Standard"*, Fifth Edition, Springer, 2003.

Chapters

Chapter No.	Title
1.	Preliminaries
~~2.~~	~~Evolution of the Major Programming Languages~~
3.	Describing Syntax and Semantics
4.	Lexical and Syntax Analysis
~~5.~~	~~Names, Binding, Type Checking and Scopes~~
6.	Data Types
7.	Expressions and Assignment Statements
8.	Statement-Level Control Structures
9.	Subprograms
10.	Implementing Subprograms
11.	Abstract Data Types and Encapsulation Constructs
12.	Support for Object-Oriented Programming
13.	Concurrency
14.	Exception Handling and Event Handling
15.	Functional Programming Languages
16.	Logic Programming Languages

Chapter 6 – Data Types

Objectives

- Introduces the concept of a data type
- Characteristics of common primitive data types
- Arrays, Records and Unions
- Pointers and References
- Design issues and the Design choices made by designers
- Implementation of various data types

Introduction

- Defines a collection of data values and a set of predefined operations
- Language should support an appropriate collection of data types and structures
- Pre-90 Fortrans, Linked Lists and Binary Trees -> Implemented with Arrays
- ALGOL 68 -> Provides a few basic types and a few flexible structure-defining operators that allow a programmer to design a data structure for each need
- User-Defined Data Types -> Structure, Union and Enumeration
- Abstract Data Types -> Linked List, Stack and Queue

Introduction

- Structured (non-scalar) Data Types -> Arrays and Records
- How Allocation and Deallocation of memory happens?
- Descriptor -> Collection of the attributes of a variable
  - Descriptor is an area of memory that stores the attributes of a variable
- Attributes are all static -> Descriptors are required only at compile time
  - Built by compiler and are used during compilation
- Dynamic attributes -> Part or all of the descriptor must be maintained during execution
  - Descriptor is used by the run-time system
- Descriptors -> Type checking + Building the code for allocation and deallocation

```
int a;
float b;
```

Variable Name	Type	Size (in Bits)
a	int	16
b	float	32

Primitive Data Types

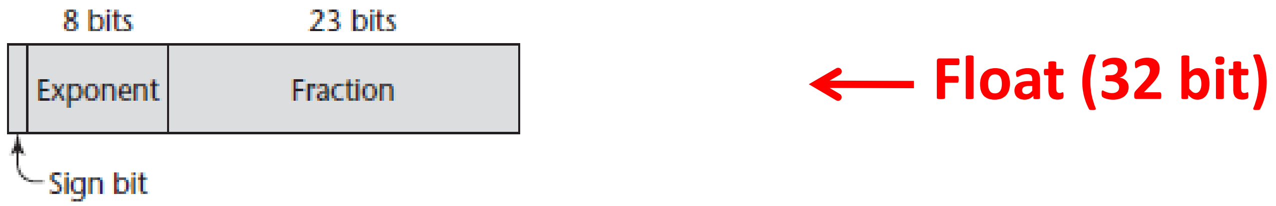
- Data types that are not defined in terms of other types are called primitive data types
- Numeric Types
  - Integer
  - Floating Point
  - Complex
  - Decimal
- Boolean Types
- Character Types

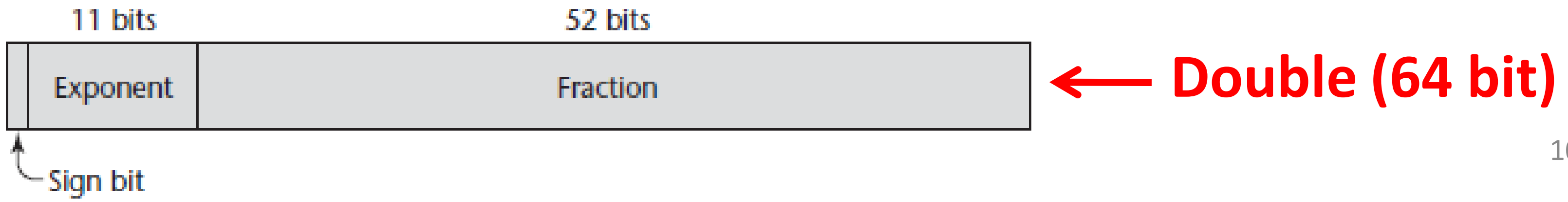
Integer Data Type

- Byte, Short, Int, Long
- Signed, Unsigned
- Signed
  - MSB = Sign, Other Bits = Value
  - Negative values are stored in 2s Complement form -> Take 1s Complement and then Add 1
- Why not 1s Complement?
  - Negative of an integer is stored as the logical complement of its absolute value. Ones-complement
  - Disadv: Has two representations of zero (+0, -0)

Floating-Point Data Type

- Represent real numbers, but the representations are only approximations
  - ∏, e
- 0.1 in decimal is 0.0001100110011 . . . in binary
- IEEE Floating-Point Standard 754 format.
  - Fractions, Exponent
- Float and Double



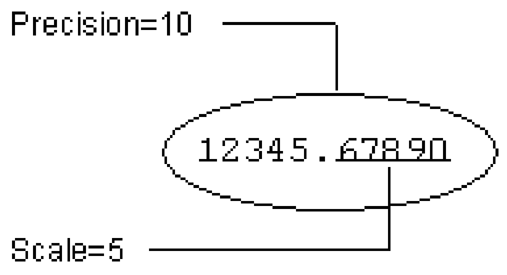


Complex Data Type

- Fortran and Python
- Represented as ordered pairs of floating-point values
- Python -> Imaginary part of a complex literal is specified by using j or J
    - (7 + 3j)
- Languages that support a complex type include operations for arithmetic on complex values

Decimal Data Type

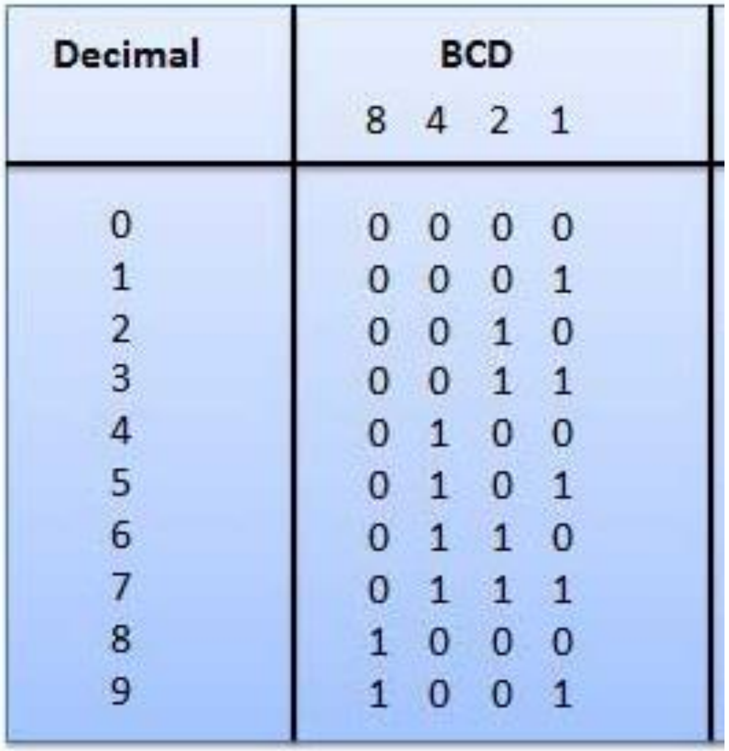
- Numeric data type defined by its precision (total number of digits) and scale (number of digits to the right of the decimal point)



- Store a fixed number of decimal digits, with the decimal point at a fixed position in the value
  - COBOL, C#, F#
- 0.1 (in decimal) can be exactly represented in a decimal type, but not in a floating-point type
- Computers that are designed to support business systems applications have hardware support for decimal data types

Decimal Data Type

- Disadv:
  - Range of values is restricted because no exponents are allowed
    - ✓ Precision -> Minimum is 1; maximum is 39
    - ✓ Scale -> Cannot exceed its precision (Can be 0)
  - Representation in memory is mildly wasteful
    - ✓ Stored very much like character strings, using binary codes for the decimal digits -> Binary Coded Decimal (BCD)
      - 456 = 010001010110 (in BCD) -> Takes more space
      - 456 = 111001000_2



Decimal	BCD			
	8	4	2	1
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1

Decimal(p, s) => Decimal(4, 2) => 10.05

100000001100101010101010101010100101010.110101001010101010

Boolean Data Type

- Simplest of all types -> True or False
- C89 -> Numeric expressions are used as conditionals
    - All operands with non-zero values -> True; 0 -> False
- Integers can be used, but the use of Boolean is more readable
- Could be represented by a single bit
- Often stored in the smallest efficiently addressable cell of memory, typically a byte

Character Types

- Stored in computers as numeric codings -> 8-bit code ASCII (American Standard Code for Information Interchange)
  - Uses the values 0 to 127 to code 128 different characters
- ISO 8859-1 is another 8-bit character code
  - Allows 256 different characters
- UCS-2 Standard, a 16-bit character set (Unicode)
  - Cyrillic alphabet -> Used in Serbia, Thai digits
- Unicode Consortium + International Standards Organization (ISO) -> 4-byte character code (UCS-4 or UTF-32)

7 (Unused)	6	5	4	3	2	1	0

Character String Types

- Values consist of sequences of characters
- Design Issues:
  - Should strings be simply a special kind of character array or a primitive type?
  - Should strings have static or dynamic length?

Strings and their Operations

- Assignment, Catenation, Substring Reference, Comparison and Pattern Matching
- Substring Reference -> Reference to a substring of a given string

  string str = “balakrishnan”

  string subStr = “krishnan”

  - Substring References are also called as Slices
- Assignment and Comparison -> Complicated with operands of different lengths
- Pattern Matching -> Specified Patterns

Strings and Operations

- C, C++ -> Use Character Arrays (Library -> string.h)
- char str[] = “apples”; -> Stored as “apples0”
- Operations: strcpy, strcmp, strlen (Does not count NULL)
- Parameters and return values for most of the string manipulation functions are char pointers that point to arrays of char
- Problems of string manipulation libraries
  - Strcpy(src, dest);   //len(src) = 50; len(dest) = 20
- C++ programmers should use string class from standard library rather than char arrays and C string library

Strings and Operations

- Pattern Matching -> Regular Expressions
- Evolved from UNIX line editor -> ed

/[A-Za-z][A-Za-z\d]+/ => a1sasdas

- Brackets enclose character classes
- First character class specifies all letters
- Second specifies all letters and digits (a digit is specified with the abbreviation \d)
- Plus operator following the second category specifies that there must be one or more of what is in the category
- Whole pattern matches strings that begin with a letter, followed by one or more letters or digits

Strings and Operations

/\d+\.?\d*|\.\d+/ => 234 or 234. or 234.5 or .343

- Pattern matches numeric literals
- \. specifies a literal decimal point
- Question mark quantifies what it follows to have zero or one appearance
- Vertical bar (|) separates two alternatives in the whole pattern
- First alternative matches strings of one or more digits, possibly followed by a decimal point, followed by zero or more digits
- Second alternative matches strings that begin with a decimal point, followed by one or more digits

• Pattern-matching capabilities using regular expressions are included in the class libraries of C++, Java, Python and C#

Memory Stack

Stack ↓ **(Return Address, Argument to Functions, Local Variables, Current CPU State)**
Heap ↑ **(Dynamic Memory Allocation – malloc, realloc, free)**
Global Variables
Program Code

String Length Options

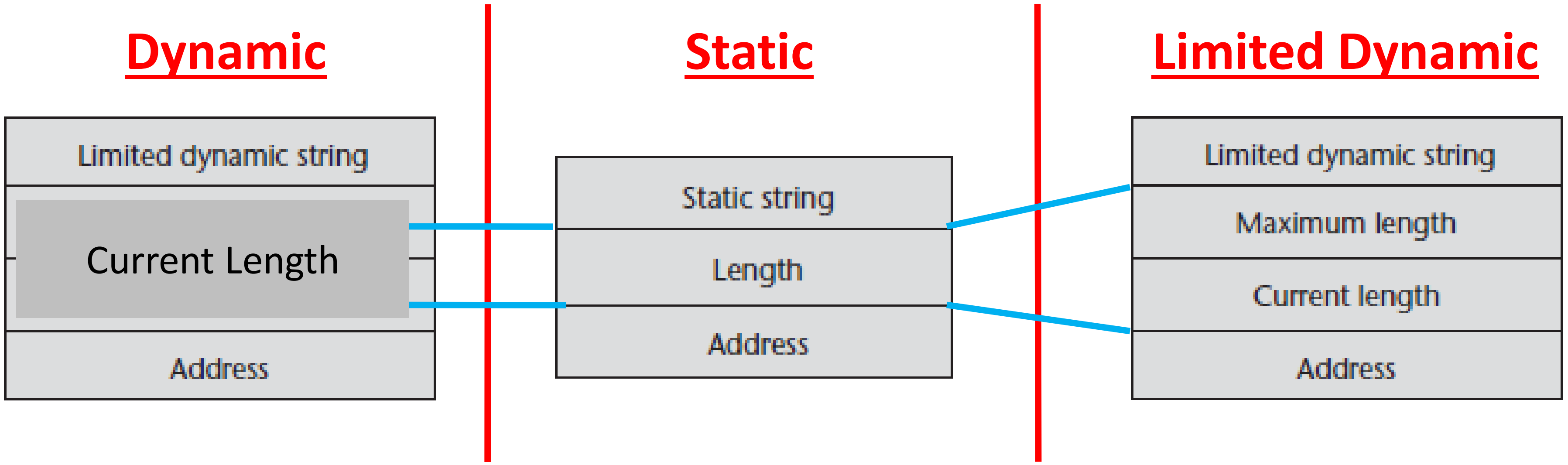
- Static -> Set when the string is created (Static String Length)
- Varying length up to a declared and fixed maximum set by variable's definition -> Limited Dynamic Length Strings
  - Can store any number of characters between zero and the maximum
- Have varying length with no maximum limit (Dynamic Length Strings)
  - overhead of dynamic storage allocation and deallocation, but provides maximum flexibility

Evaluation

- String types are important to the writability of a language
- Dealing with strings as arrays is not efficient than dealing with a primitive string type
  - strcpy in C -> Simple assignment statement would require a loop
- Providing strings through a standard library is nearly as convenient as having them as a primitive type
- String operations such as simple pattern matching and catenation are essential and should be included for string type values

Implementation of Character String Types

- A descriptor for a static character string type, which is required only during compilation, has three fields



- Limited dynamic strings require a run-time descriptor to store both the fixed maximum length and the current length
- Dynamic length strings require a simpler run-time descriptor because only the current length needs to be stored

Miscellaneous

```
int a;
float b;
```

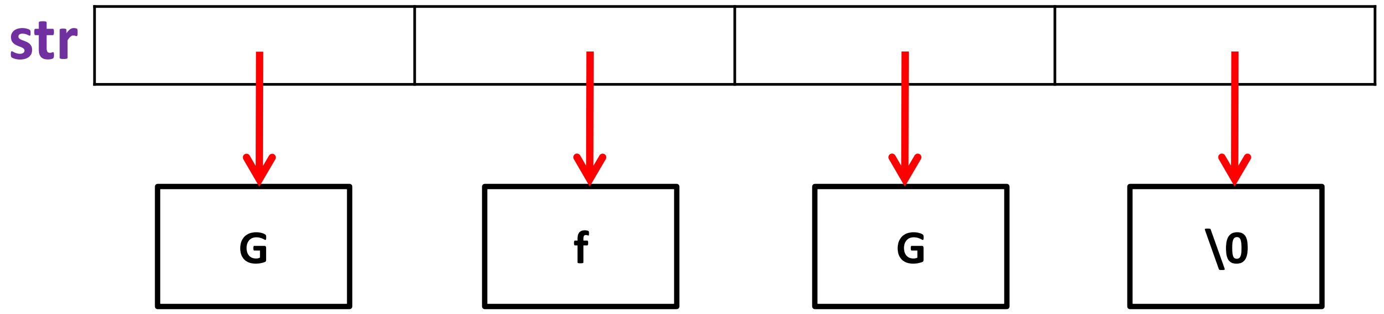
Variable Name	Type	Size (in Bits)
a	int	16
b	float	32

Implementation of Character String Types

- Limited dynamic strings of C and C++
  - Do not require run-time descriptors -> Null Character
  - Do not need the maximum length -> Index values in array references are not range-checked
- Dynamic length strings require more complex storage management -> Grow and Shrink dynamically
- Three approaches
  - Strings can be stored in a linked list -> Heap memory
    - ✓ Disadv: Extra memory + Complexity in String Operations
  - Store strings as arrays of pointers to individual characters allocated in Heap
    - ✓ Extra memory + Faster String Operations

Implementation of Character String Types

```
char *str;
int size = 4; /*one extra for ‘\0’*/
str = (char *)malloc(sizeof(char)*size);
*(str+0) = 'G';
*(str+1) = 'f';
*(str+2) = 'G';
*(str+3) = '\0';
```



Implementation of Character String Types

- Store complete strings in adjacent storage cells
  - New area of memory is found that can store the complete new string
- Discussion
  - Linked List -> Requires more storage, allocation and deallocation processes are simple
    - ✓ Disadv: String operations are slowed by the required pointer chasing
  - Adjacent memory cells -> Faster string operations, Less storage
    - ✓ Disadv: Allocation and deallocation processes are slower

Array Types

- Homogeneous aggregate of data elements in which an individual element is identified by its position in the aggregate, relative to the first element

a	25	35	45	55	65
	0	1	2	3	4

Address of a[4] = Address of a[0] + (4 * size of Data Type in Bytes)

- Individual data elements are of the same type
- References to individual array elements are specified using subscript expressions
- If any of the subscript expressions in a reference include variables, then the reference will require an additional run-time calculation to determine the address of the memory location being referenced

x = a[5 + y]

Design Issues

- What types are legal for subscripts?
- Are subscripting expressions in element references range checked?
- When are subscript ranges bound?
- When does array allocation take place?
- Are ragged or rectangular multidimensioned arrays allowed, or both?
- Can arrays be initialized when they have their storage allocated?
- What kinds of slices are allowed, if any?

Arrays and Indices

- Specific elements are referenced by means of a two-level syntactic mechanism -> Name + Subscripts/Indices
  - If all of the subscripts in a reference are constants, the selector is static
- Selection operation can be thought of as a mapping from the array name and the set of subscript values to an element in the aggregate -> Finite Mappings

array_name(subscript_value_list)

- In Ada, Sum := sum + B(I); -> Reduces Readability
  - Array element references map the subscripts to a particular element
  - Function calls map the actual parameters to the function definition and, eventually, a functional value

Arrays and Indices

- Two distinct types are involved in an array type: the element type and the type of the subscripts

a[x]

```
type Week_Day_Type is (Monday, Tuesday, Wednesday,
                       Thursday, Friday);
type Sales is array (Week_Day_Type) of Float;
```

Sales(Monday) = 6.55; => Sales(0) = 6.55;

- Early programming languages did not specify that subscript ranges must be implicitly checked
  - C, C++, Perl -> Do not
- One can reference an array element in Perl with a negative subscript, Eg: $list[-2]

Subscript Bindings and Array Categories

- Binding of the subscript type to an array variable is usually static, but the range of values are sometimes dynamically bound
- C -> Lower bound = 0; Fortran 95 -> Lower bound = 1
- Five categories of arrays -> Binding to Subscript ranges, Binding to storage, From where the storage is allocated
  - Static Array -> Subscript ranges are statically bound and storage allocation is static (done before run time)
    - ✓ Adv: Efficiency + No dynamic allocation or deallocation is required
    - ✓ Disadv: Storage for the array is fixed for the entire execution time of the program

Subscript Bindings and Array Categories

- Fixed stack-dynamic array -> Subscript ranges are statically bound, but the allocation is done at declaration elaboration time during execution
  - ✓ Adv: Space efficiency -> A large array in one subprogram can use the same space as a large array in a different subprogram, as long as both subprograms are not active at the same time. The same is true if the two arrays are in different blocks that are not active at the same time
  - ✓ Disadv: Required allocation and deallocation time
- Stack-dynamic array -> Both subscript ranges and the storage allocation are dynamically bound at elaboration time
  - ✓ Once the values are fixed, they remain unchanged during the lifetime of the variable
  - ✓ Adv: Flexibility-> Size need not be known until the array is about to be used

Subscript Bindings and Array Categories

- Fixed-heap dynamic array -> Similar to a fixed stack-dynamic array with few differences
  - ✓ Subscript ranges and the storage binding are both fixed after storage is allocated
  - ✓ Subscript ranges and storage bindings are done when the user program requests them during execution
  - ✓ Storage is allocated from heap rather than from the stack
  - ✓ Adv: Array’s size always fits the problem
  - ✓ Disadv: Allocation time is longer
- Heap-dynamic array -> Binding of subscript ranges and storage allocation is dynamic and can change any number of times during the array’s lifetime
  - ✓ Adv: Flexibility -> Grow or shrink
  - ✓ Disadv: Allocation and deallocation take longer and may happen many times during execution of the program

Subscript Bindings and Array Categories

- Arrays declared in C and C++ functions that include the static modifier are static

- Arrays that are declared in C and C++ functions (without the static specifier) are examples of fixed stack-dynamic arrays

- C and C++ also provide fixed heap-dynamic arrays
  - Standard C library functions malloc and free, which are general heap allocation and deallocation operations, respectively, can be used for C arrays

- Heterogeneous Arrays -> Elements need not be of the same type
  - Perl, Python, JS, Ruby

Array Initialization

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

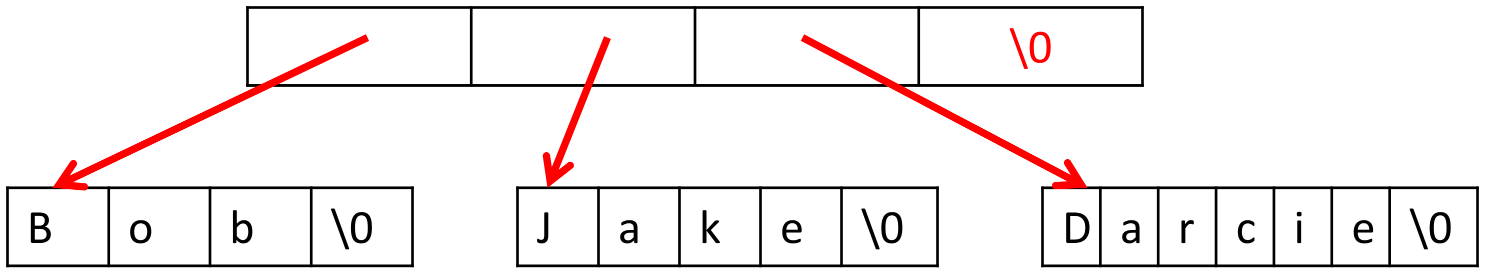
- Compiler sets the length of the array

```
char name [] = “freddie”;
```

- Character strings are implemented as arrays of **char**
- Have eight elements, because all strings are terminated with a null character (zero)

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

- Array is one of pointers to characters



Array Operations

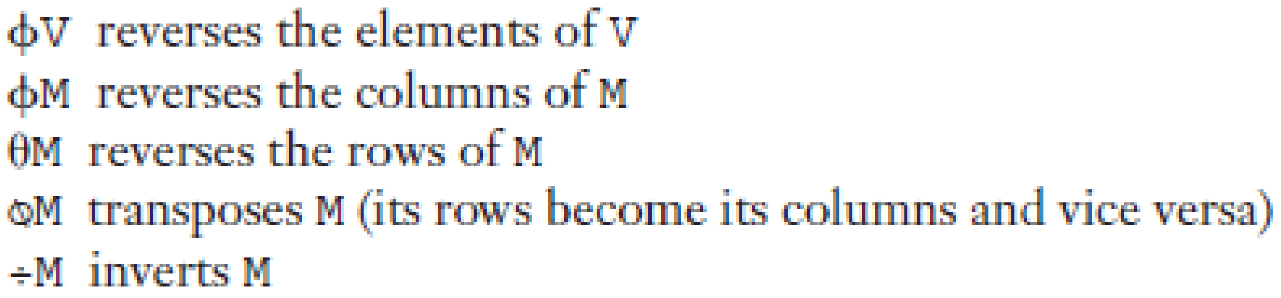
- An array operation is one that operates on an array as a unit
- Common operations -> Assignment, Catenation, Comparison for equality and inequality, Slices
- APL
  - Arrays and their operations are the heart of APL
  - It is the most powerful array-processing language ever devised

Array Operations

- In APL, the four basic arithmetic operations are defined for vectors (single-dimensioned arrays) and matrices, as well as scalar operands

A + B

- Is a valid expression, whether A and B are scalar variables, vectors, or matrices
- APL includes a collection of unary operators for vectors and matrices (where V is a vector and M is a matrix):



ϕV reverses the elements of V
ϕM reverses the columns of M
θM reverses the rows of M
⍉M transposes M (its rows become its columns and vice versa)
÷M inverts M

Array Operations

- V = [1 2 3 4 5 6] ; M = [1 2 3
  4 5 6
  7 8 9]

- ϕV = [6 5 4 3 2 1]
- ϕM = [7 8 9
  4 5 6
  1 2 3]

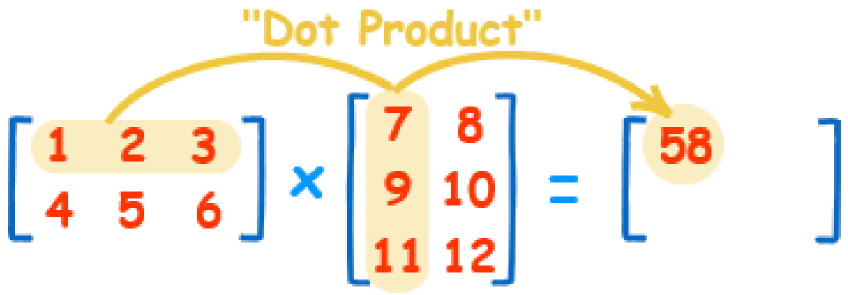
- θM = [3 2 1
  6 5 4
  9 8 7]
- ⍉M = [1 4 7
  2 5 8
  3 6 9]

Array Operations

- If A and B are vectors, A × B is the mathematical inner product of A and B (a vector of the products of the corresponding elements of A and B)

  A = [1 2 3]; B = [2 2 2]; A x B = [2 4 6]

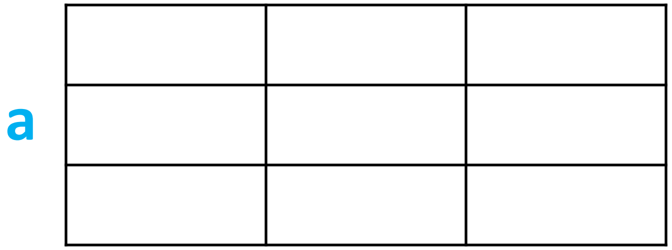
- APL includes several special operators that take other operators as Operands
- One of these is the inner product operator, which is specified with a period (.)
  - Takes two operands, which are binary operators, Eg: +.×
  - New operator that takes two arguments, either vectors or matrices
  - First multiplies the corresponding elements of two arguments, and then it sums the Results
  - A +.× B -> Matrix Multiplication



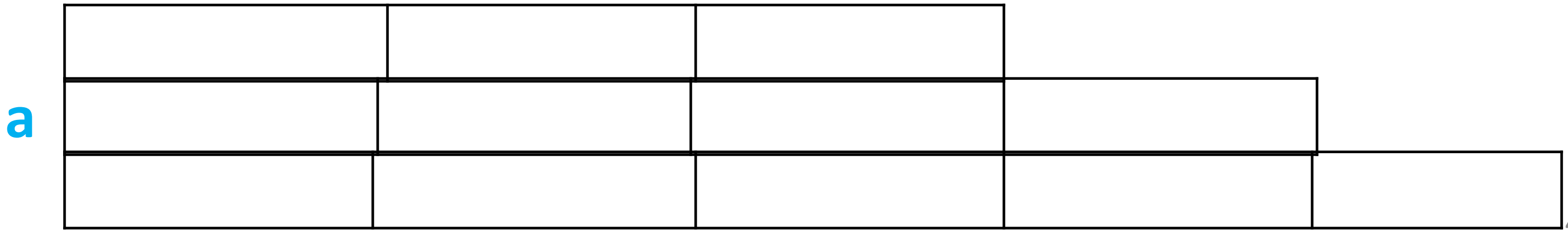
$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix} = \begin{bmatrix} 58 & \end{bmatrix}$$

Rectangular and Jagged Arrays

- Rectangular Array -> Multidimensioned array
  - All of the rows have the same number of elements
  - All of the columns have the same number of elements
  - Rectangular arrays model rectangular tables exactly



- C, C++, Java -> a[2][3]
- Fortran, Ada and C# -> a[2, 3]
- Jagged Array -> Lengths of the rows need not be the same



Slices

- Substructure of that array
- It is important to realize that a slice is not a new data type
- Mechanism for referencing part of an array as a unit
- If arrays cannot be manipulated as units in a language, that language has no use for slices

```
vector = [2, 4, 6, 8, 10, 12, 14, 16]
mat = [[1, 2, 3],[4, 5, 6],[7, 8, 9]]
```

- vector[3:6] -> Three-element array with the fourth through sixth elements of vector
- mat[1] -> Second row of mat
- mat[0][0:2] -> First and Second element of the first row of mat, which is [1, 2]

Implementation of Array Types

- Implementing arrays requires considerably more compile-time effort than does implementing primitive types
- Code to allow accessing of array elements must be generated at compile time
- At run time, this code must be executed to produce element addresses
- No way to precompute the address to be accessed by a reference, list[k]

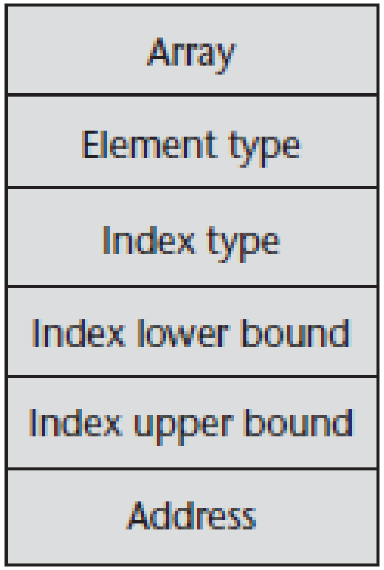
address(list[k]) = address(list[0]) + k * element_size

- If the element type is statically bound and the array is statically bound to storage, then the value of the constant part can be computed before run time

```
address(list [k]) = address(list[lower_bound]) +
                        ((k - lower_bound) * element_size)
```

Implementation of Array Types

- Compile-time descriptor for single-dimensioned arrays includes information required to construct the access function
- If run-time checking of index ranges is not done and the attributes are all static, then only the access function is required during execution; no descriptor is needed
- If run-time checking of index ranges is done, then those index ranges may need to be stored in a run-time descriptor
- If the subscript ranges of a particular array type are static, then the ranges may be incorporated into the code that does the checking, thus eliminating the need for the run-time descriptor
- If any of the descriptor entries are dynamically bound, then those parts of the descriptor must be maintained at run time



Implementation of Array Types

- Two ways in which multidimensional arrays can be mapped to one dimension: row major order and column major order

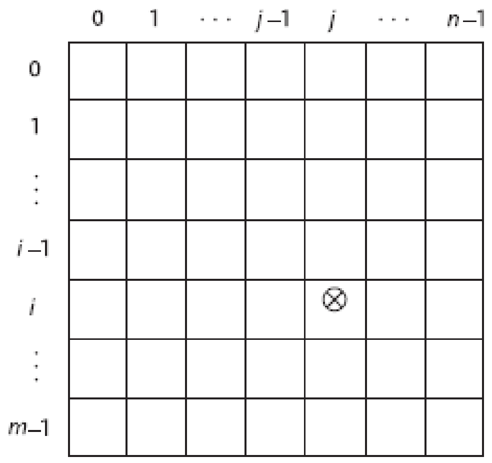
3 4 7

6 2 5

1 3 8

- Row Major Order -> 3, 4, 7, 6, 2, 5, 1, 3, 8
- Column Major Order (Fortran) -> 3, 6, 1, 4, 2, 3, 7, 5, 8

Implementation of Array Types



(((i – row_lb) * n) + (j – col_lb)) * element_size
= (i * n * element_size) – (row_lb * n * element_size) +
(j * element_size) – (col_lb * element_size)

location(a[i, j]) = address of a[0, 0]
+ ((((number of rows above the *ith* row) * (size of a row))
+ (number of elements left of the *jth* column)) *
element size)

location(a[i, j]) = $\underline{\text{address of a[0, 0]}}$ + $\underline{\text{(((i * n) + j) *}}$
$\underline{\text{element\_size)}}$

location(a[i, j]) = address of a[row_lb, col_lb]
+ (((i – row_lb) * n) + (j – col_lb)) * element_size

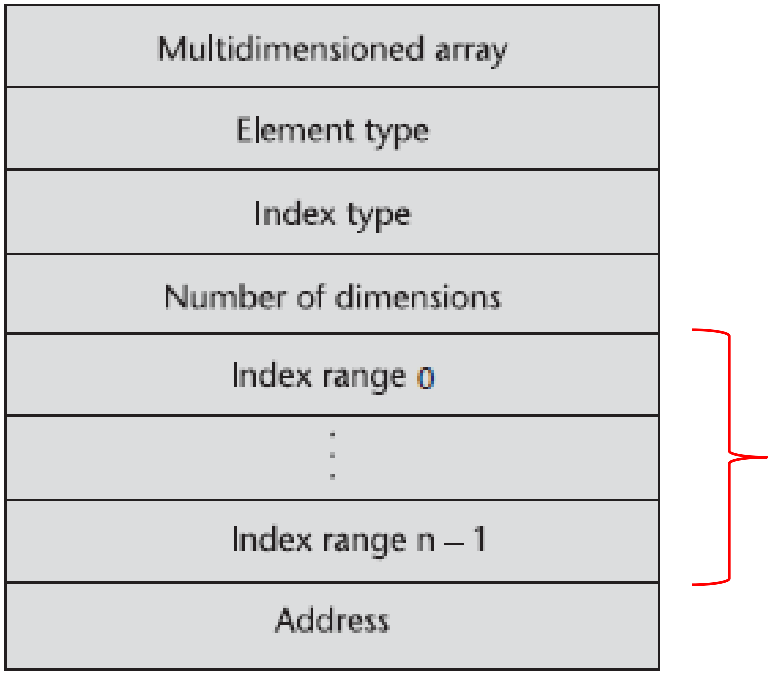
location(a[i, j]) = $\underline{\text{address of a[row\_lb, col\_lb]}}$
$\underline{\text{– (((row\_lb * n) + col\_lb) * element\_size)}}$
$\underline{\text{+ (((i * n) + j) * element\_size)}}$

Implementation of Array Types

- For each dimension of an array, one add and one multiply instruction are required for the access function
- Accesses to elements of arrays with several subscripts are costly

$$\text{address}(\texttt{list [k]}) = \text{address}(\texttt{list}[\text{lower\_bound}]) + ((\texttt{k} - \text{lower\_bound}) * \text{element\_size})$$

$$\text{location}(\texttt{a [i, j]}) = \text{address of a}[\text{row\_lb}, \text{col\_lb}] + (((\texttt{i} - \text{row\_lb}) * \text{n}) + (\texttt{j} - \text{col\_lb})) * \text{element\_size}$$



Associative Arrays

- Unordered collection of data elements that are indexed by an equal number of values called keys

Key	Value
0	100
2	20
4	30
1	30
3	10

- Non-associative arrays, the indices never need to be stored (because of their regularity)

a	100	30	20	10	30
	0	1	2	3	4

Structure and Operations

Perl:

```
%salaries = ("Gary" => 75000, "Perry" => 57000,
             "Mary" => 55750, "Cedric" => 47850);
```

```
$salaries{"Perry"} = 58850;
```

```
delete $salaries{"Gary"};
```

```
@salaries = ();
```

```
if (exists $salaries{"Shelly"}) . . .
```

Structure and Operations

- *keys* operator -> Returns an array of the keys
- *values* operator -> Returns an array of the values
- *each* operator -> Iterates over the element pairs
- Hash is much efficient
  - If searches of the elements are required
  - Data to be stored is paired

Name	Salary
Bala	10,000/-
Krishnan	20,000/-

- If every element of the list is to be processed, it is more efficient to use an array

Implementing Associative Arrays

- Perl
  - 32-bit hash value is computed for each entry and is stored with the entry
  - When an associative array must be expanded beyond its initial size, the hash function need not be changed; rather, more bits of the hash value are used
  - Only half of the entries must be moved when this happens
  - So, although expansion of an associative array is not free, it is not as costly as might be expected
- PHP
  - Elements in arrays are placed in memory through a hash function
  - All elements are linked together in the order in which they were created
  - Links are used to support iterative access to elements through the current and next functions

Record Types

- Aggregate of data elements in which the individual elements are identified by names and accessed through offsets from the beginning of the structure
    - Information about a college student -> Name, #, GPA

```
struct Books
{
   char title[50];
   char author[50];
   char subject[100];
   int book_id;
} book;
```

Record Types

- Design Issues
  - What is the syntactic form of references to fields
  - Are elliptical references allowed?

Definitions of Records

Sl. No.	Records	Arrays
1.	Heterogenous	Homogenous
2.	References are made using identifiers	References are made using indices
3.	Allowed to include Unions	Not possible

EMPLOYEE-RECORD record consists of the EMPLOYEE-NAME record and the HOURLY-RATE field. The numerals 01, 02, and 05 that begin the lines of the record declaration are level numbers, which indicate by their relative values the hierarchical structure of the record. Any line that is followed by a line with a higher-level number is itself a record.

COBOL

```
01  EMPLOYEE-RECORD.
    02  EMPLOYEE-NAME.
        05  FIRST    PICTURE IS X(20).
        05  MIDDLE   PICTURE IS X(10).
        05  LAST     PICTURE IS X(20).
    02  HOURLY-RATE PICTURE IS 99V99.
```

Ada

```
type Employee_Name_Type is record
   First  : String (1..20);
   Middle : String (1..10);
   Last  : String (1..20);
end record;
type Employee_Record_Type is record
   Employee_Name: Employee_Name_Type;
   Hourly_Rate: Float;
end record;
Employee_Record: Employee_Record_Type;
```

References to Record Fields

COBOL

```
field_name OF record_name_1 OF . . . OF record_name_n
```

```
MIDDLE OF EMPLOYEE-NAME OF EMPLOYEE-RECORD
```

Other

```
Employee_Record.Employee_Name.Middle
```

- Fully Qualified Reference -> All intermediate record names from the largest enclosing record to the specific field are named in the reference
- Elliptical Reference -> Any or all of the enclosing record names can be omitted, as long as the resulting reference is unambiguous
  - FIRST, FIRST OF EMPLOYEE-NAME, and FIRST OF EMPLOYEE-RECORD are elliptical references

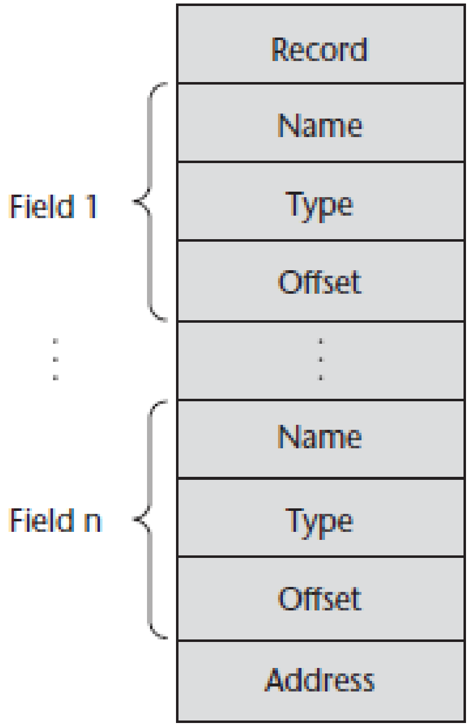
```
01  EMPLOYEE-RECORD.
    02  EMPLOYEE-NAME.
        05  FIRST     PICTURE IS X(20).
        05  MIDDLE    PICTURE IS X(10).
        05  LAST      PICTURE IS X(20).
    02  HOURLY-RATE PICTURE IS 99V99.
```

Implementation of Record Types

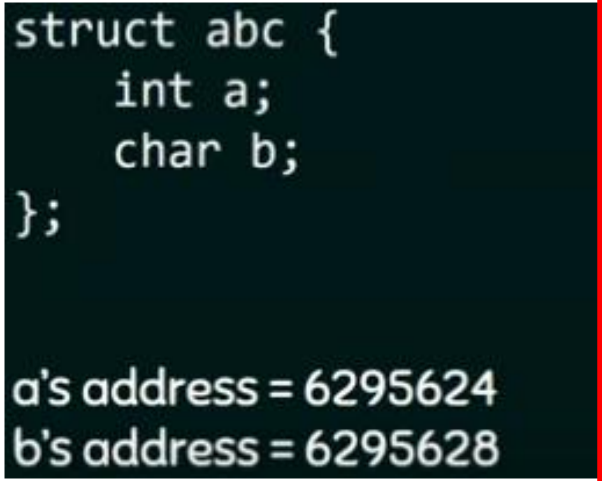
- Fields of records are stored in adjacent memory locations
- Access method used for arrays cannot be used for records
  - Offset address, relative to the beginning of the record, is associated with each field

```
struct Books
{
   char title[50];
   char author[50];
   char subject[100];
   int book_id;
} book;
```

- Field accesses are all handled using these offsets

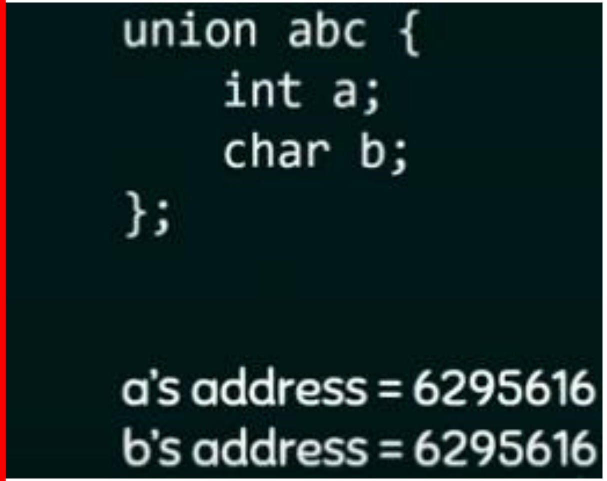


Struct vs Union



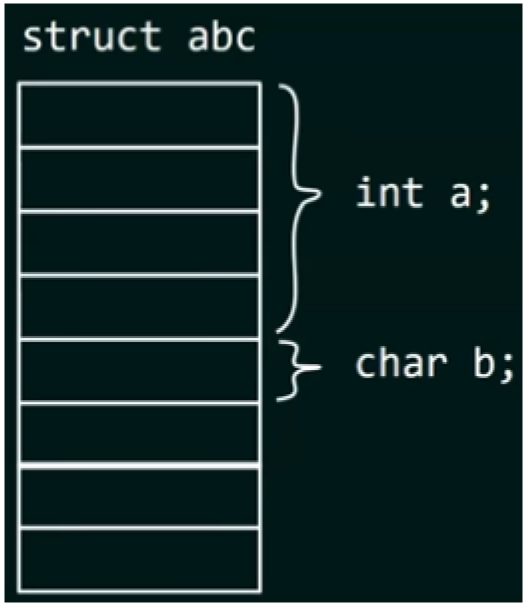
```
struct abc {
    int a;
    char b;
};
```

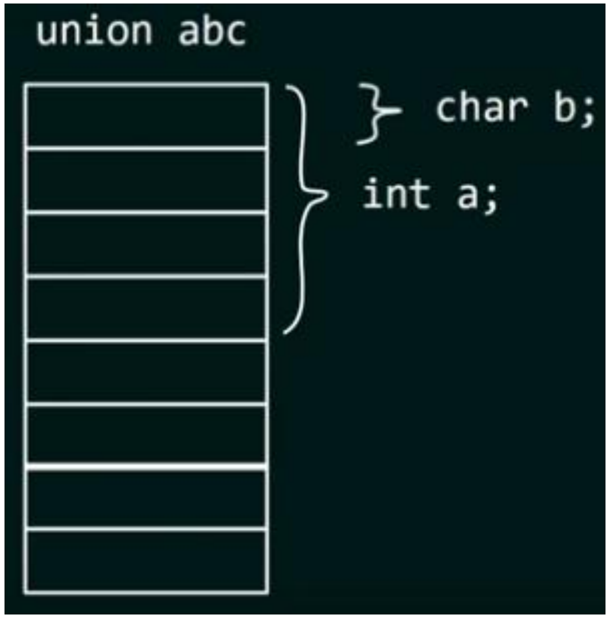
a's address = 6295624
b's address = 6295628



```
union abc {
    int a;
    char b;
};
```

a's address = 6295616
b's address = 6295616





Union Types

- A type whose variables may store different type values at different times during program execution

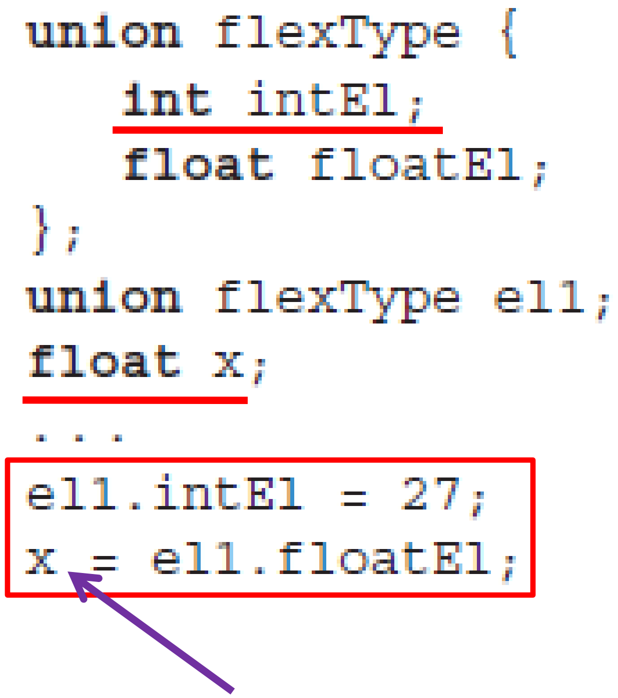
```
union Data
{
   int i;              //2 Bytes
   float f;            //4 Bytes
   char str[20];       //20 Bytes
} data;
```

- Share memory
- Required size is 20 Bytes

- Design issues
  - Should type checking be required?
  - Should unions be embedded in records?

Discriminated vs Free Unions

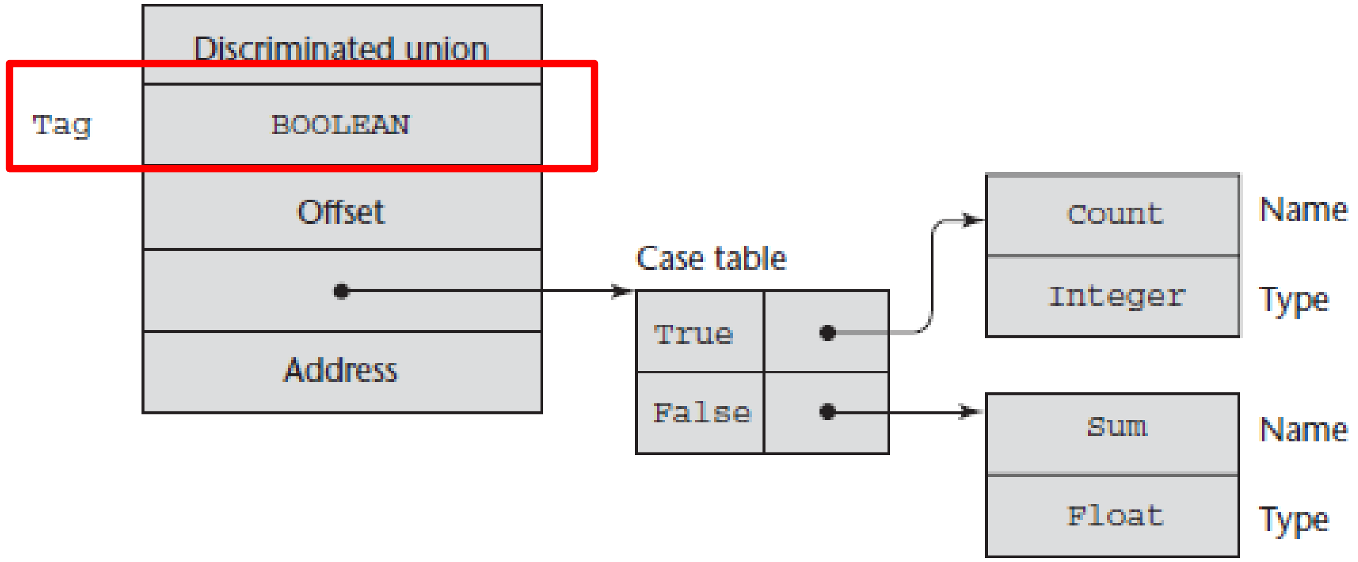
- Fortran, C, C++ -> No type checking
  - Free Unions
- Last assignment is not type checked
  - System cannot determine the current type of the current value of ell
  - Assigns the bit string representation of 27 to the float variable x



```
union flexType {
    int intEl;
    float floatEl;
};
union flexType el1;
float x;
. . .
el1.intEl = 27;
x = el1.floatEl;
```

Discriminated vs Free Union

- Type checking of unions requires that each union construct include a type indicator
- Such an indicator is called a tag or discriminant
- Union with a discriminant is called a discriminated union
- First language to support is ALGOL68 -> Later Ada



Ada Union Types

- Constrained Variant Variable -> Allows to specify variables of a variant record type that will store only one of the possible type values in the variant
    - Type checking can be static
- Unconstrained Variant Variable -> Allow the values of their variants to change types during execution
    - Type of the variant can be changed only by assigning the entire record, including the discriminant

Ada Union Types

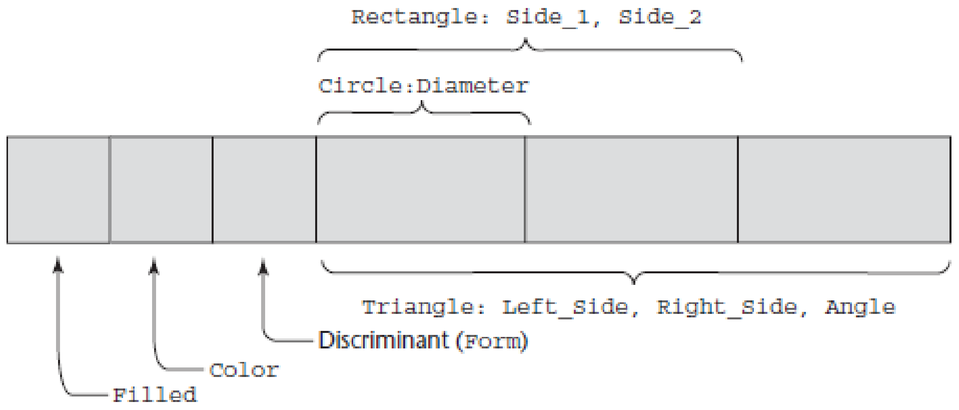
```
type Shape is (Circle, Triangle, Rectangle);
type Colors is (Red, Green, Blue);
type Figure (Form : Shape) is
  record
    Filled : Boolean;
    Color : Colors;
    case Form is
      when Circle =>
        Diameter : Float;
      when Triangle =>
        Left_Side : Integer;
        Right_Side : Integer;
        Angle : Float;
      when Rectangle =>
        Side_1 : Integer;
        Side_2 : Integer;
    end case;
  end record;
```

```
Figure_1 : Figure;
Figure_2 : Figure(Form => Triangle);
```

Figure_1 is declared to be an unconstrained variant record that has no initial value. Its type can change by assignment of a whole record, including the discriminant

```
Figure_1 := (Filled => True,
             Color => Blue,
             Form => Rectangle,
             Side_1 => 12,
             Side_2 => 3);
```

```
if(Figure_1.Diameter > 3.0) ...
```



Rectangle: Side_1, Side_2
Circle:Diameter
Triangle: Left_Side, Right_Side, Angle
Discriminant (Form)
Color
Filled

Evaluation

- Unions are potentially unsafe constructs in some languages like C and C++

    - Not strongly typed -> These languages do not allow type checking of references to their unions

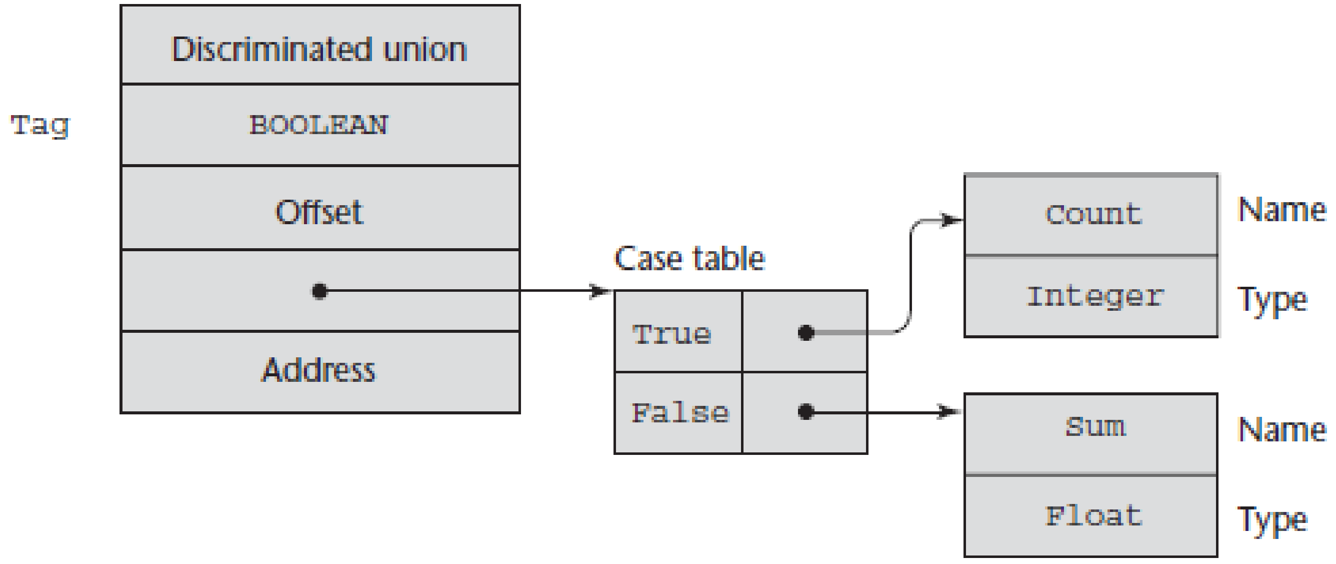
- Unions can be safely used in Ada

Implementation

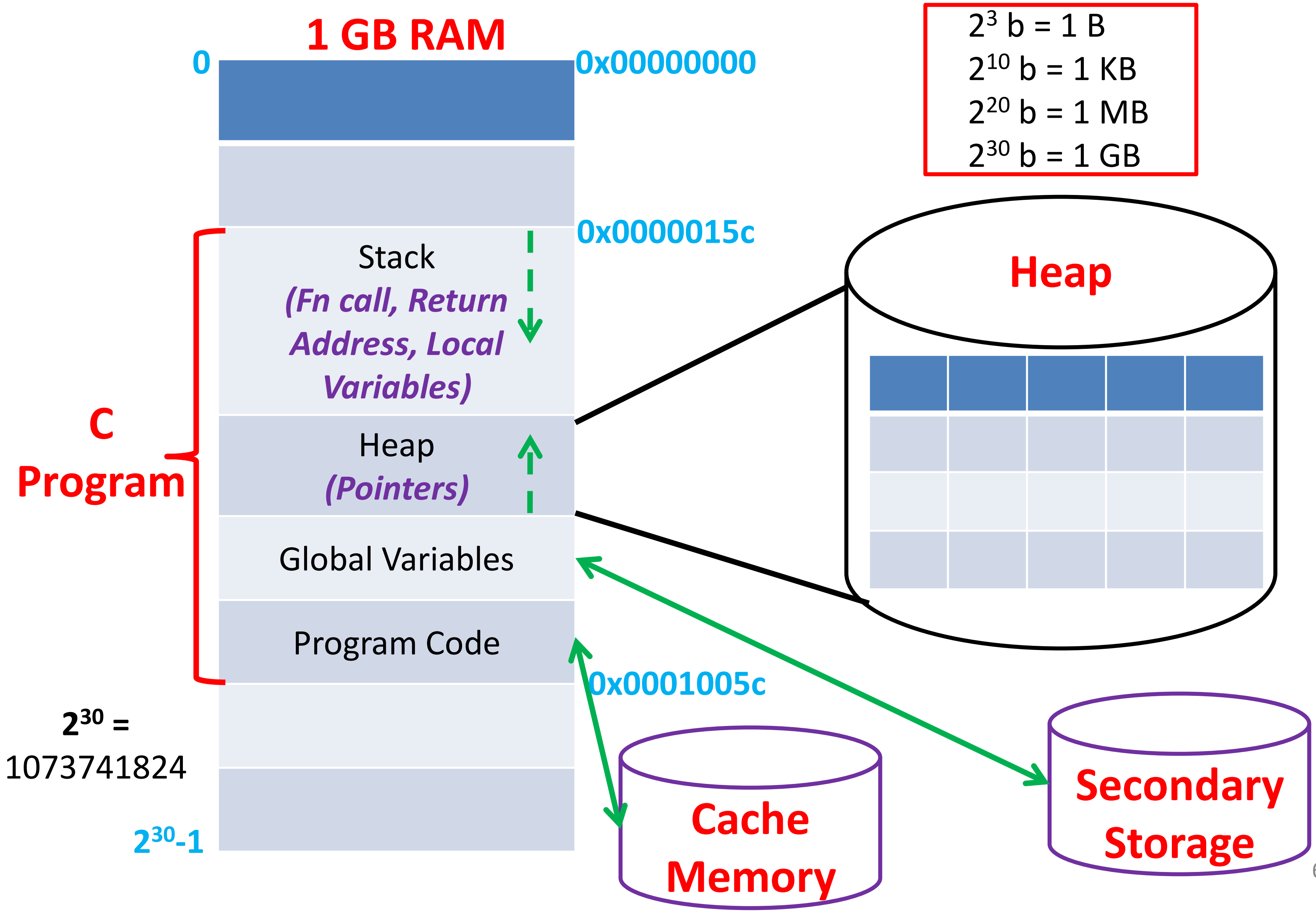
- Unions are implemented by simply using the same address for every possible variant
- Sufficient storage for the largest variant is allocated
- Tag of a discriminated union is stored with the variant in a record-like structure
- At compile time, the complete description of each variant must be stored
    - Can be done by associating a case table with the tag entry in the descriptor
    - Case table has an entry for each variant, which points to a descriptor for that particular variant

Implementation

```
type Node (Tag : Boolean) is
  record
  case Tag is
      when True => Count : Integer;
      when False => Sum : Float;
  end case;
 end record;
```



Memory Management



2^3 b = 1 B
2^{10} b = 1 KB
2^{20} b = 1 MB
2^{30} b = 1 GB

Binary to Hex

$00\ 0000\ 1010\ 0010_2$

$0000\ 0000\ 1010\ 0010_2$

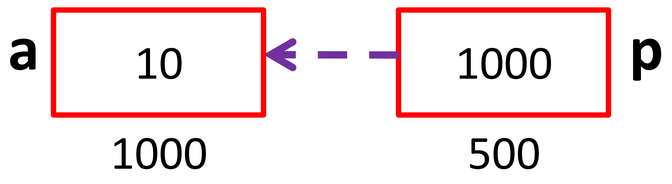
$0 \quad 0 \quad A \quad 2_{16}$

0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Pointer vs Reference

- Pointer Variable

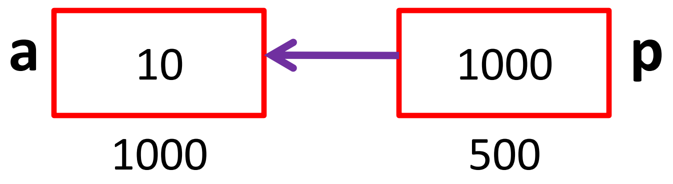
```
int a = 10;
int *p = &a;

int a = 10;
int *p;
p = &a;
```



Initialization
Reassignment
NULL Value
Indirection
Arithmetic Operations

- Reference Variable

```
int a=10;
int & p=a;
```



Pointer vs Reference

Pass-by-Value

```
#include <iostream>
using namespace std;
void square(int &);
int main()
{
  int number = 8;
  cout << "In main(): " << &number << endl;        // 0x22ff1c
  cout << number << endl;                           // 8
  square(number);
  cout << number << endl;                           // 8
}

void square(int rNumber)                 // New Local variable rNumber will be created
{
  cout << "In square(): " << &rNumber << endl;     // 0x22ff1f
  rNumber *= rNumber;
}
```

Pointer vs Reference

Pass-by-Reference with Pointer Arguments

```
#include <iostream>
using namespace std;
void square(int *);
int main()
{
   int number = 8;
   cout << "In main(): " << &number << endl;          // 0x22ff1c
   cout << number << endl;                             // 8
   square(&number);                           // Explicit referencing to pass an address
   cout << number << endl;                            // 64
}
void square(int * pNumber)                            // Function takes an int pointer
{
   cout << "In square(): " << pNumber << endl;        // 0x22ff1c
   *pNumber *= *pNumber;        // Explicit de-referencing to get the value pointed-to
}
```

Pointer vs Reference

Pass-by-Reference with Reference Arguments

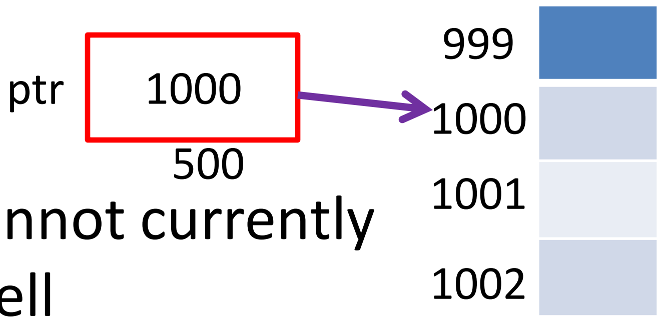
```
#include <iostream>
using namespace std;
void square(int &);
int main()
{
   int number = 8;
   cout << "In main(): " << &number << endl;        // 0x22ff1c
   cout << number << endl;                          // 8
   square(number);                                  // Implicit referencing (without '&')
   cout << number << endl;                          // 64
}

void square(int & rNumber)                          // Function takes an int reference
{
   cout << "In square(): " << &rNumber << endl;     // 0x22ff1c
   rNumber *= rNumber;                              // Implicit de-referencing (without '*')
}
```

Static vs Dynamic Memory Allocation

```
// Static Allocation
int number = 88;
int * p1 = &number;              // Assign a "valid" address into pointer

// Dynamic Allocation
int * p2;             // Not initialize, points to somewhere which is invalid
cout << p2 << endl;              // Print address before allocation
p2 = new int;  // Dynamically allocate an int and assign its address to pointer
                      // The pointer gets a valid address with memory allocated
*p2 = 99;
cout << p2 << endl;              // Print address after allocation
cout << *p2 << endl;            // Print value point-to
delete p2;                      // Remove the dynamically allocated storage
```

Pointer and Reference Types

- A pointer type is one in which the variables have a range of values that consists of memory addresses and a special value, NIL (NULL/0)
- NIL is not a valid address
  - Used to indicate that a pointer cannot currently be used to reference a memory cell



- Purpose
  - Indirect addressing
  - Provide a way to manage dynamic storage. It can be used to access a location in an area where storage is dynamically allocated called a heap

Pointer and Reference Types

- Variables that are dynamically allocated from the heap are called heap-dynamic variables
- They often do not have identifiers associated with them
    - Can be referenced only by pointer or reference type variable
- Variables without names -> Anonymous Variables
- Pointers, unlike arrays and records, are not structured types
- Also different from scalar variables -> Used to reference some other variable (Reference Type) rather than being used to store data (Value Type)
- Use of pointers add writability to a language

Design Issues

- What are the scope and lifetime of a pointer variable?
- What is the lifetime of a heap-dynamic variable (the value a pointer references)?
- Are pointers restricted as to the type of value to which they can point?
- Are pointers used for dynamic storage management, indirect addressing, or both?
- Should the language support pointer types, reference types, or both?

Pointer Operations

- Operations: Assignment and Dereferencing

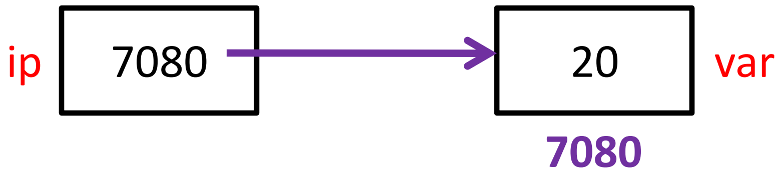
```
int var = 20;
int *ip;
ip = &var;
int x = *ip;
```

- Assignment Operation -> Sets a pointer variable's value to some useful address
  - If pointer variables are used only to manage dynamic storage, then the allocation mechanism, whether by operator or built-in subprogram, serves to initialize the pointer variable
  - If pointers are used for indirect addressing to variables that are not heap dynamic, then there must be an explicit operator or built-in subprogram for fetching the address of a variable, which can then be assigned to the pointer variable

Pointer Operations

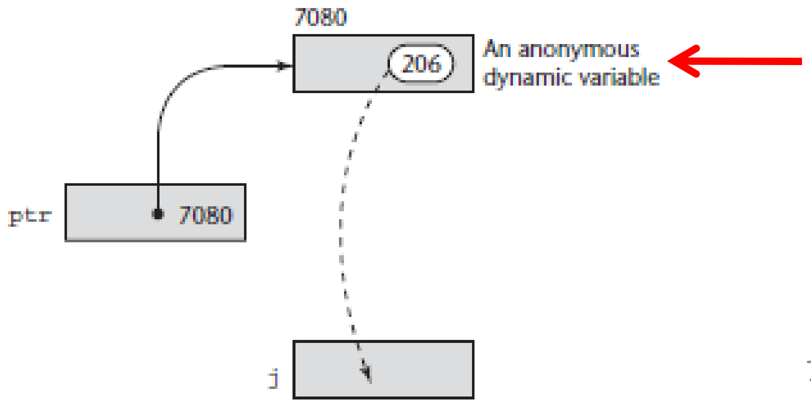
- An occurrence of a pointer variable in an expression can be interpreted in two distinct ways
  - Reference to the contents of the memory cell to which it is bound, which in the case of a pointer is an address -> Normal Reference Pointer

```
int var = 20;
int *ip;
ip = &var;
```



  - Reference to the value in the memory cell pointed to by the memory cell to which the pointer variable is bound. In this case, the pointer is interpreted as an indirect reference -> Dereferencing the Pointer

```
j = *ptr // Sets j = 206
```



Pointer Operations

- When pointers point to records, the syntax varies among languages
- C, C++ -> Two ways a pointer to a record can be used to reference a field in that record
  - p → age; (*p).age
- Languages that provide pointers for the management of a heap must include an explicit allocation and deallocation operation
  - malloc, new
  - delete

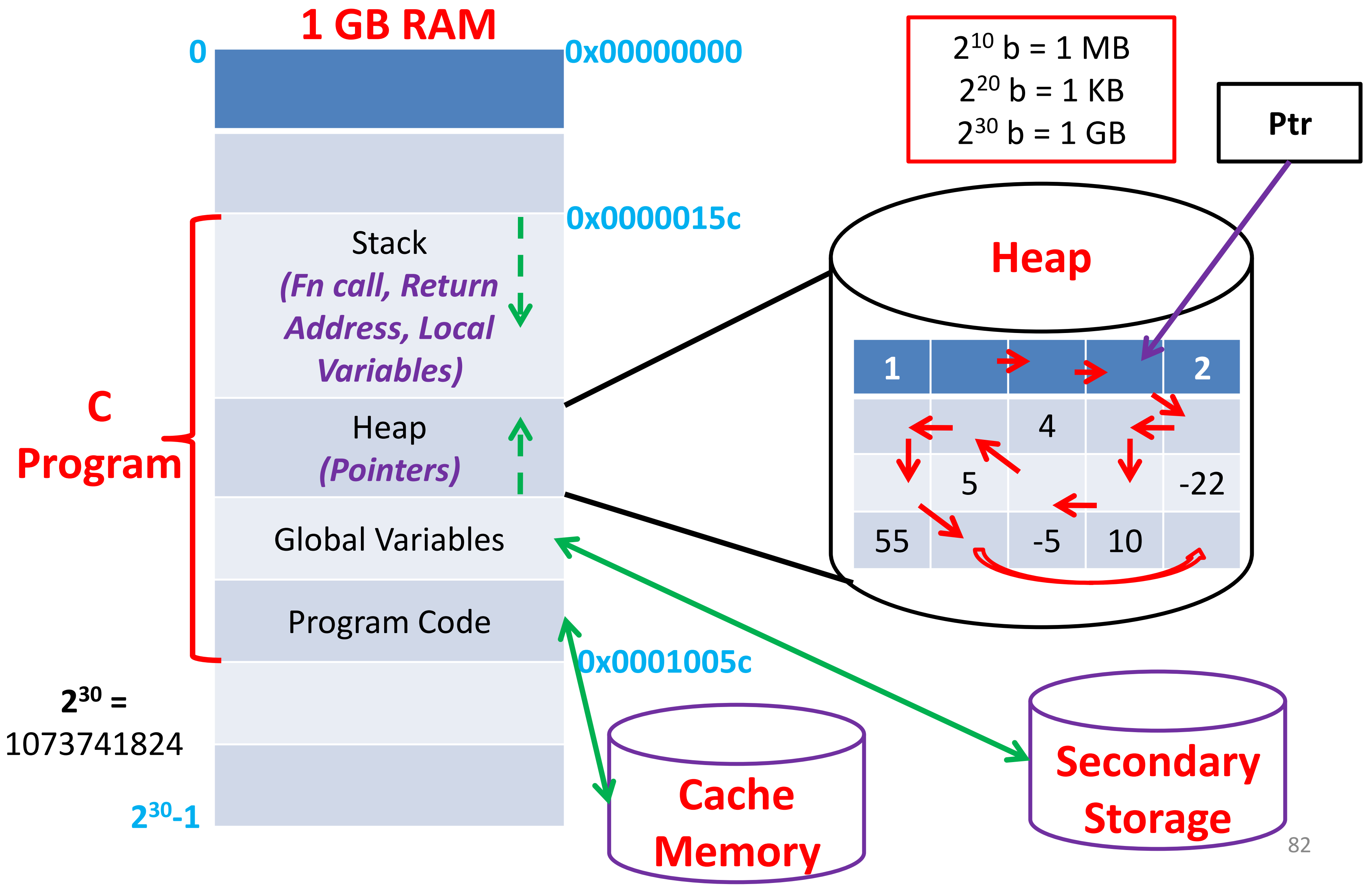
Pointer Problems

- First high-level programming language to include pointer variables was PL/I
  - Pointers could be used to refer to both heap-dynamic variables and other program variables
  - Were highly flexible, but their use could lead to several kinds of programming errors
- Some of the problems of PL/I pointers are also present in the pointers of subsequent languages
- Recent languages, such as Java, have replaced pointers completely with reference types
- A reference type is really only a pointer with restricted operations

Dangling Pointers

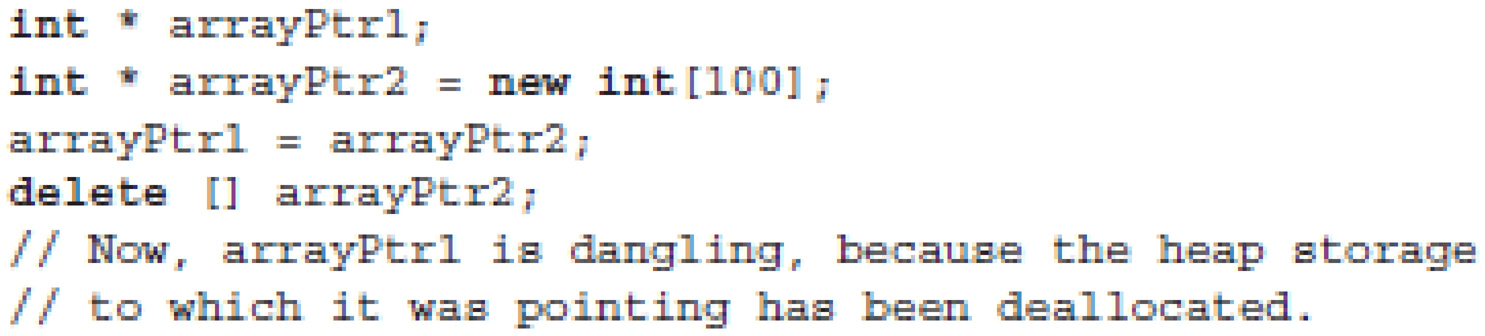
- Also called as dangling reference
- Pointer that contains the address of a heap-dynamic variable that has been deallocated
- Problems
    - Location being pointed to may have been reallocated to some new heap-dynamic variable
    - If the new variable is not the same type as the old one, type checks of uses of the dangling pointer are invalid
    - Even if the new dynamic variable is the same type, its new value will have no relationship to the old pointer's dereferenced value
    - Furthermore, if the dangling pointer is used to change the heap-dynamic variable, the value of the new heap-dynamic variable will be destroyed
    - Finally, it is possible that the location now is being temporarily used by the storage management system, possibly as a pointer in a chain of available blocks of storage, thereby allowing a change to the location to cause the storage manager to fail

Memory Management



Dangling Pointers

- Possible ways to create Dangling Pointers
  - A new heap-dynamic variable is created and pointer p2 is set to point at it
  - Pointer p1 is assigned p2's value
  - The heap-dynamic variable pointed to by p2 is explicitly deallocated (possibly setting p2 to nil), but p1 is not changed by the operation
  - p1 is now a dangling pointer
  - If the deallocation operation did not change p2, both p1 and p2 would be dangling. (Of course, this is a problem of aliasing—p1 and p2 are aliases)



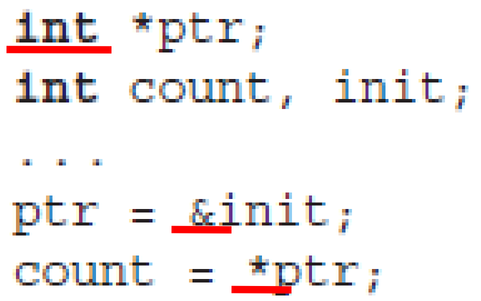
```
int * arrayPtr1;
int * arrayPtr2 = new int[100];
arrayPtr1 = arrayPtr2;
delete [] arrayPtr2;
// Now, arrayPtr1 is dangling, because the heap storage
// to which it was pointing has been deallocated.
```

Lost Heap-Dynamic Variables

- An allocated heap-dynamic variable that is no longer accessible to the user program
- Such variables are often called garbage, because they are not useful for their original purpose, and they also cannot be reallocated for some new use in the program
- Possible way to create
  - Pointer p1 is set to point to a newly created heap-dynamic variable
  - p1 is later set to point to another newly created heap-dynamic variable
- The first heap-dynamic variable is now inaccessible or lost -> Memory Leakage
- Memory leakage is a problem, regardless of whether the language uses implicit or explicit deallocation

Pointers in C and C++

- Can point anywhere in memory, whether there is a variable there or not, which is one of the dangers of such pointers
- Asterisk (*) denotes the dereferencing operation
- Ampersand (&) denotes the operator for producing the address of a variable



```
int *ptr;
int count, init;
. . .
ptr = &init;
count = *ptr;
```

count = init;

- Pointers can be assigned the address value of any variable of the correct domain type, or they can be assigned the constant zero, which is used for nil.

Pointers in C and C++

ptr + index

- Instead of simply adding the value of index to ptr, the value of index is first scaled by the size of the memory cell (in memory units) to which ptr is pointing (its base type)

```
int list [10];
int *ptr;
ptr = list;   // Assigns the address of list[0] to ptr
```

- *(ptr + 1) is equivalent to list[1]
- *(ptr + index) is equivalent to list[index]
- ptr[index] is equivalent to list[index]
- Pointers in C and C++ can point to functions
    - Used to pass functions as parameters to other functions

Pointers in C and C++

- C and C++ include pointers of type void *, which can point at values of any type -> Generic Pointers
    - Type checking is not a problem with void * pointers, because these languages disallow dereferencing them

```
int a = 10;       void *ptr = &a;       printf("%d", *ptr);
// printf("%d", *(int *)ptr);
```

```
int a = 10;
char b = 'x';
void *p = &a;
p = &b;
```

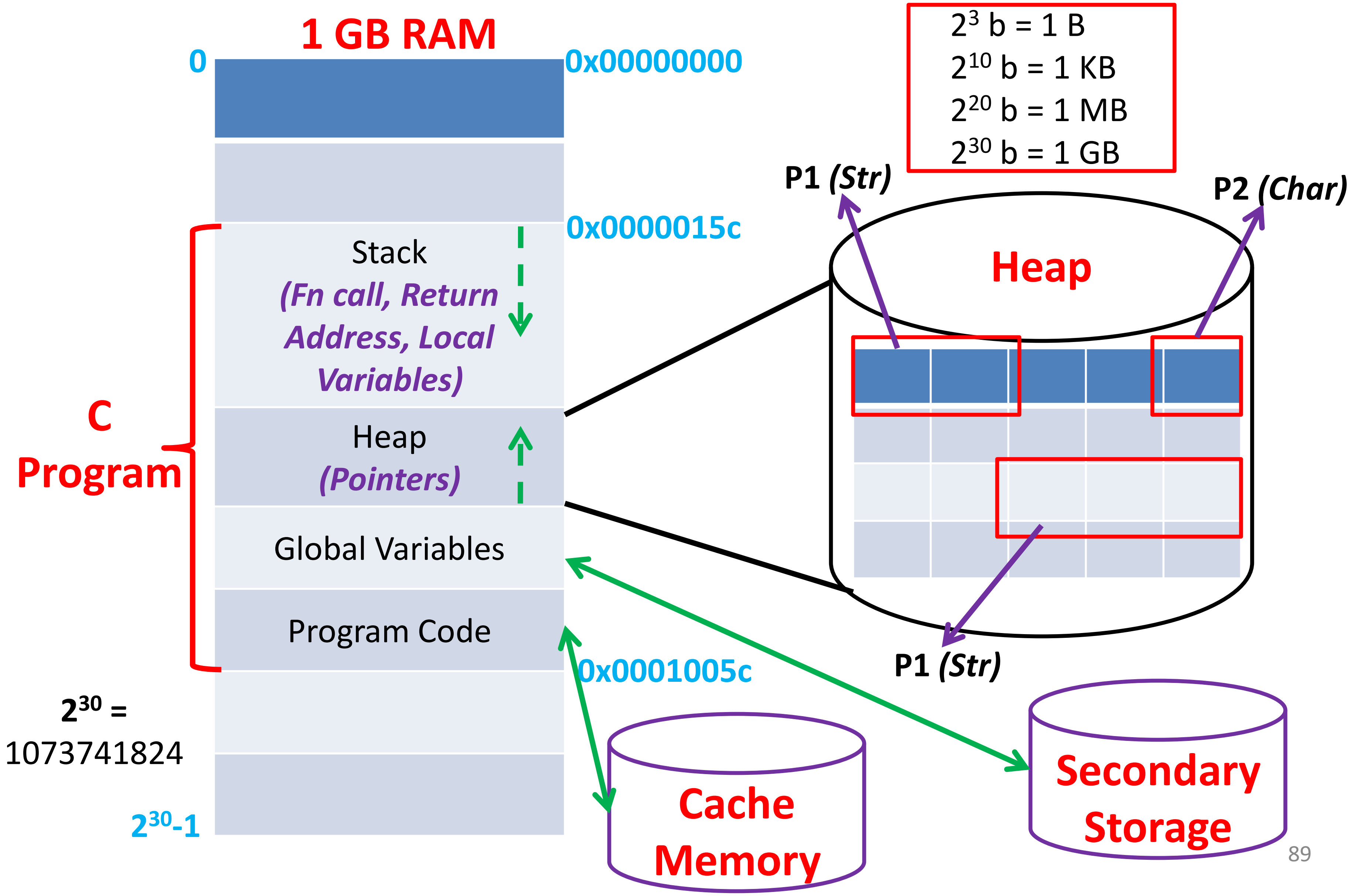
```
int main(void)
{
   // Note that malloc() returns void * which can be
   // typecasted to any type like int *, char *, ..
   int *x = malloc(sizeof(int) * n);
}
```

Pointers in C and C++

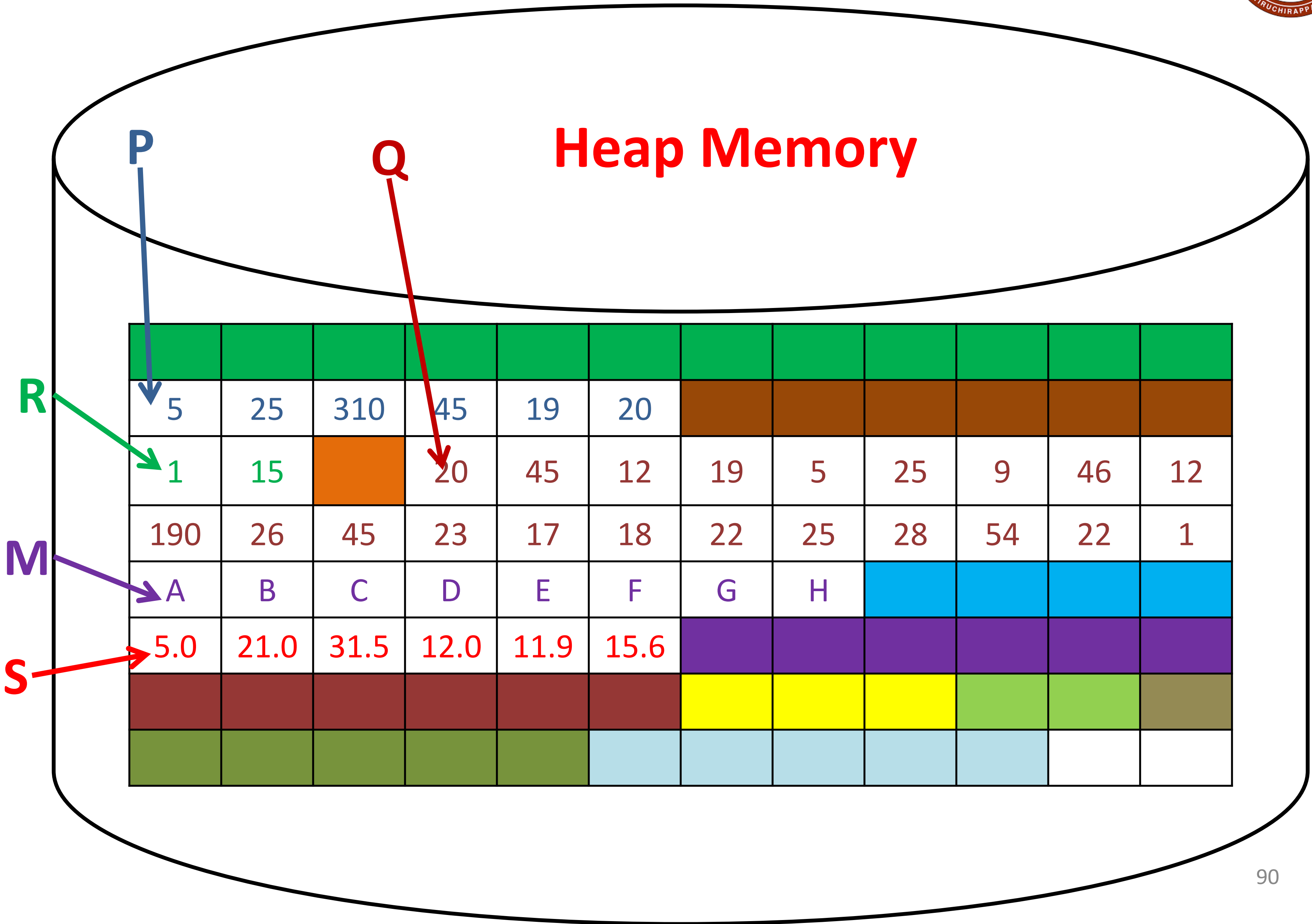
- Common use of void * pointers is as the types of parameters of functions that operate on memory
- Eg: Suppose we wanted a function to move a sequence of bytes of data from one place in memory to another
- It would be most general if it could be passed two pointers of any type
- This would be legal if the corresponding formal parameters in the function were void * type
- The function could then convert them to char * type and do the operation, regardless of what type pointers were sent as actual parameters

Memory Management

2^3 b = 1 B
2^{10} b = 1 KB
2^{20} b = 1 MB
2^{30} b = 1 GB



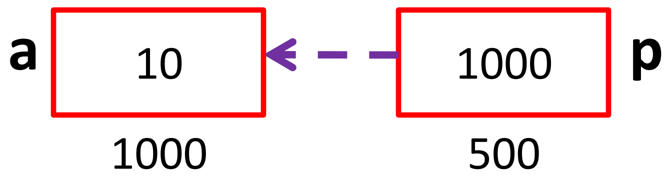
Miscellaneous



Pointer vs Reference

- Pointer Variable

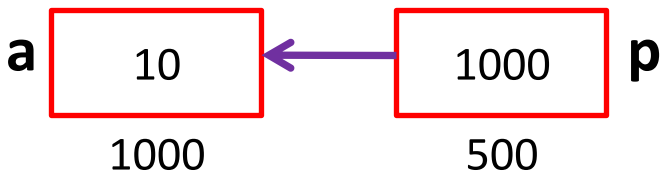
```
int a = 10;
int *p = &a;

int a = 10;
int *p;
p = &a;
int j = p;
int j = *p;
```



Initialization
Reassignment
NULL Value
Indirection
Arithmetic Operations

- Reference Variable

```
int a=10;
int & p=a;
int j = p;
```



Reference Types

- A pointer refers to an address in memory, while a reference refers to an object or a value in memory
- Although it is natural to perform arithmetic on addresses, it is not sensible to do arithmetic on references

  ptr + index // Valid for Pointers and not for References

- C++ includes a special kind of reference type that is:
  - Used primarily for the formal parameters in function definitions
  - A constant pointer and is always implicitly dereferenced
  - Must be initialized with the address of some variable in its definition
  - After initialization a reference type variable can never be set to reference any other variable
- The implicit dereference prevents assignment to the address value of a reference variable

```
int result = 0;
int &ref_result = result;
. . .
ref_result = 100;
```

result and ref_result are aliases

Evaluation

- Problems: Dangling Pointers; Garbage
- Pointers have been compared with the goto
  - Goto statement widens the range of statements that can be executed next
  - Pointer variables widen the range of memory cells that can be referenced by a variable
- Application: Pointers are necessary to write device drivers, in which specific absolute addresses must be accessed

Implementation of Pointer and Reference Types

- Pointers are used in heap management
  - First, describe how pointers and references are represented internally
- Discuss two possible solutions to the dangling pointer problem
- Finally, describe the major problems with heap-management techniques

Representations of Pointers and References

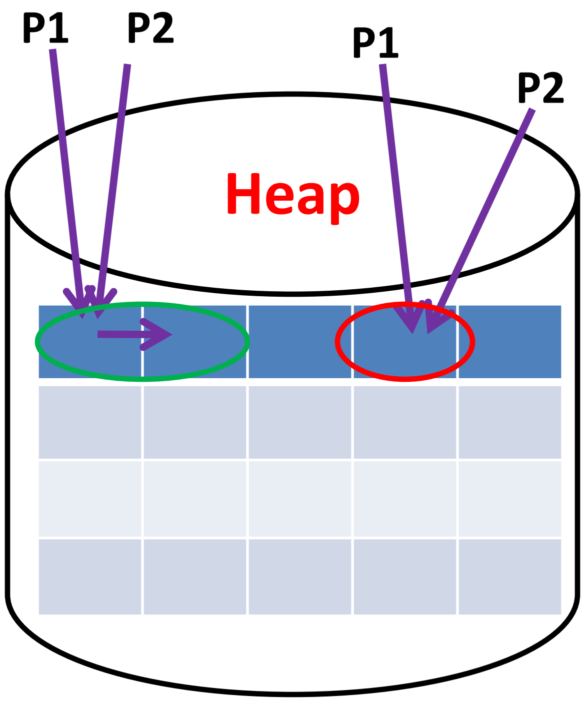
- Pointers and References are single values stored in memory cells

- Intel microprocessors, addresses have two parts: a segment and an offset

  - Pointers and References are implemented in these systems as pairs of 16-bit cells, one for each of the two parts of an address

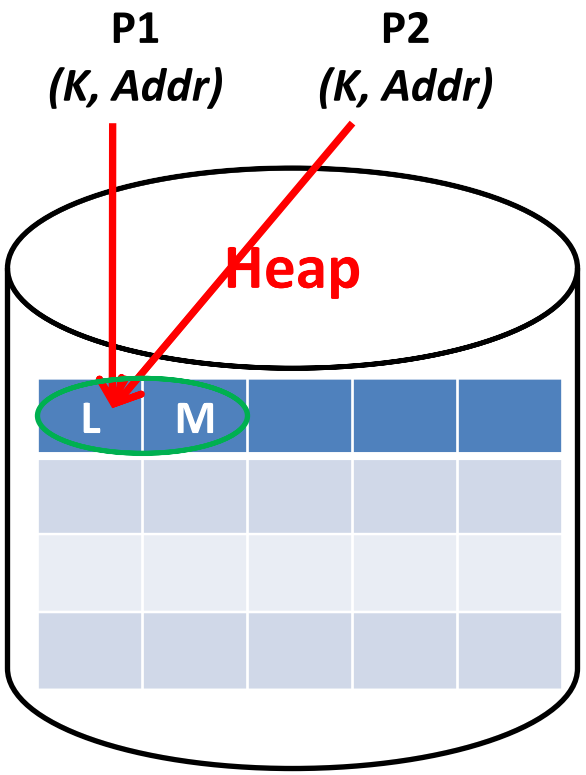
Solutions to the Dangling-Pointer Problem

- **Tombstones**
  - Every heap-dynamic variable includes a special cell, called a tombstone, that is itself a pointer to the heap-dynamic variable
  - Actual pointer variable points only at tombstones and never to heap-dynamic variables
  - When a heap-dynamic variable is deallocated, the tombstone remains but is set to nil, indicating that the heap-dynamic variable no longer exists -> Approach prevents a pointer from ever pointing to a deallocated variable
  - Any reference to any pointer that points to a nil tombstone can be detected as an error
  - Costly in both time and space
  - Their storage is never reclaimed
  - Every access to a heap dynamic variable through a tombstone requires one more level of indirection



Solutions to the Dangling-Pointer Problem

- **Lock-and-Keys Approach**
  - Pointer Values -> Ordered Pairs (Key, Address)
  - Heap-dynamic variables -> Storage for the variable + a header cell (stores an integer lock value)
  - When a heap-dynamic variable is allocated, a lock value is created and placed both in the lock cell of the heap-dynamic variable and in the key cell of the pointer that is specified in the call to new
  - Every access to the dereferenced pointer compares the key value of the pointer to the lock value in the heap-dynamic variable
  - If they match, the access is legal
  - Otherwise, the access is treated as a run-time error



Solutions to the Dangling-Pointer Problem

- Any copies of the pointer value to other pointers must copy the key value
- Therefore, any number of pointers can reference a given heap dynamic variable
- When a heap-dynamic variable is deallocated with dispose, its lock value is cleared to an illegal lock value
- Then, if a pointer other than the one specified in the dispose is dereferenced, its address value will still be intact, but its key value will no longer match the lock, so the access will not be allowed

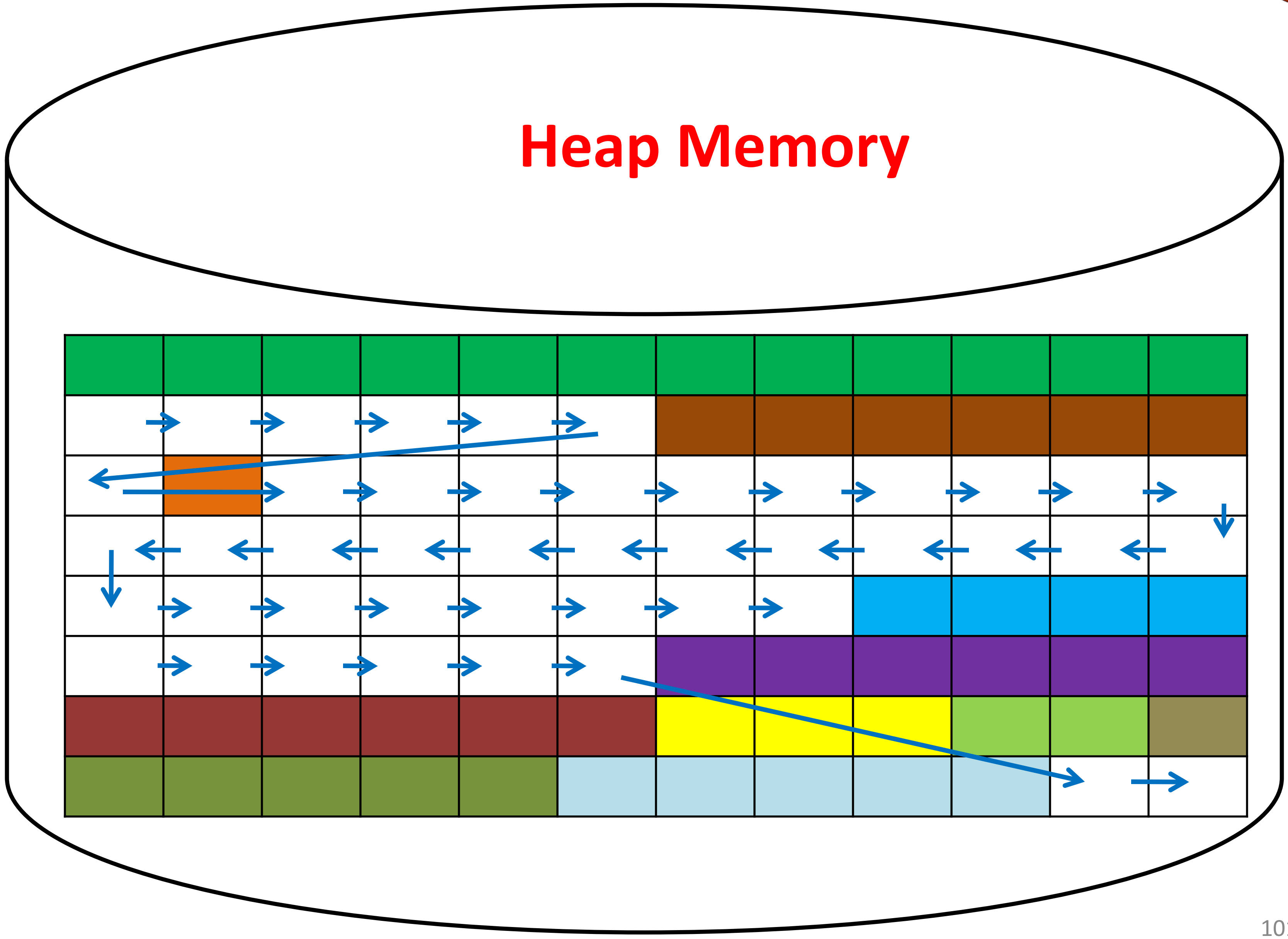
Solutions to the Dangling-Pointer Problem

- Best solution to the dangling-pointer problem is to take deallocation of heap-dynamic variables out of the hands of programmers
- If programs cannot explicitly deallocate heap-dynamic variables, there will be no dangling pointers
- To do this, the run-time system must implicitly deallocate heap-dynamic variables when they are no longer useful

Heap Management

- Complex run-time process
- Examine in two separate situations
  - One in which all heap storage is allocated and deallocated in units of a single size
  - One in which variable-size segments are allocated and deallocated

Single-Size Cells



Single-Size Cells

- In a single-size allocation heap, all available cells are linked together using the pointers in the cells, forming a list of available space
- Allocation is a simple matter of taking the required number of cells from this list when they are needed
- A heap-dynamic variable can be pointed to by more than one pointer, making it difficult to determine when the variable is no longer useful to the program
- Simply because one pointer is disconnected from a cell obviously does not make it garbage
- There could be several other pointers still pointing to the cell

Single-Size Cells

- Several different approaches to garbage collection
    - Reference Counters -> Reclamation is incremental and is done when inaccessible cells are created (Eager Approach)
    - Mark-Sweep -> Reclamation occurs only when the list of available space becomes empty (Lazy Approach)

Reference Counter Method

- Maintains in every cell a counter that stores the number of pointers that are currently pointing at the cell
- Embedded in the decrement operation for the reference counters, which occurs when a pointer is disconnected from the cell, is a check for a zero value
- If the reference counter reaches zero, it means that no program pointers are pointing at the cell, and it has thus become garbage and can be returned to the list of available space

Reference Counter Method

- Problems
  - If storage cells are relatively small, the space required for the counters is significant
  - Some execution time is obviously required to maintain the counter values
  - Every time a pointer value is changed, the cell to which it was pointing must have its counter decremented, and the cell to which it is now pointing must have its counter incremented
    - Solution: Deferred Reference Counting -> Avoids reference counters for some pointers
  - Complications arise when a collection of cells is connected circularly
    - Each cell in the circular list has a reference counter value of at least 1, which prevents it from being collected and placed back on the list of available space

Reference Counter Method

- Advantages
  - It is intrinsically incremental
  - Its actions are interleaved with those of the application, so it never causes significant delays in the execution of the application

Mark-and-Sweep Process

- Run-time system allocates storage cells as requested and disconnects pointers from cells as necessary, without regard for storage reclamation (allowing garbage to accumulate), until it has allocated all available cells

- At this point, a mark-sweep process is begun to gather all the garbage left floating around in the heap

- To facilitate the process, every heap cell has an extra indicator bit or field that is used by the collection algorithm

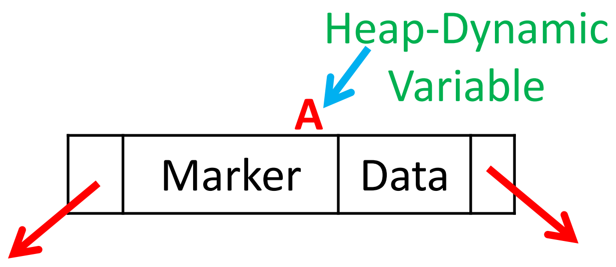
Mark-and-Sweep Process

- Consists of three distinct phases
  - First Phase -> All cells in the heap have their indicators set to indicate they are garbage
  - Marking Phase -> Every pointer in the program is traced into the heap and all reachable cells are marked as not being garbage
  - Sweep Phase -> All cells in the heap that have not been specifically marked as still being used, are returned to the list of available space

Mark-and-Sweep Process

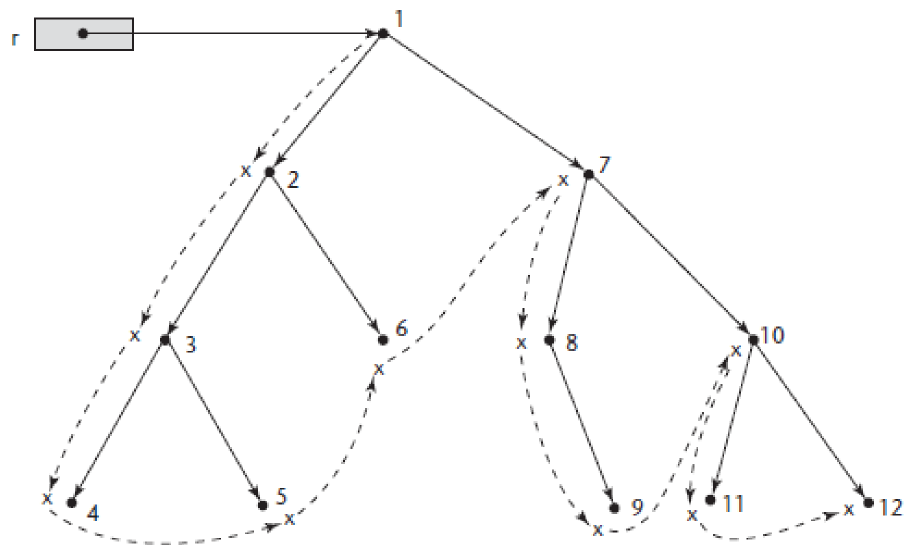
- Assume that all heap-dynamic variables, or heap cells, consist of an information part; a part for the mark, named marker; and two pointers named llink and rlink
- These cells are used to build directed graphs with at most two edges leading from any node
- The marking algorithm traverses all spanning trees of the graphs, marking all cells that are found
- Like other graph traversals, the marking algorithm uses recursion

```
for every pointer r do
    mark(r)

void mark(void * ptr) {
    if (ptr != 0)
      if (*ptr.marker is not marked) {
        set *ptr.marker
        mark(*ptr.llink)
        mark(*ptr.rlink)
      }
}
```



Mark-and-Sweep Process

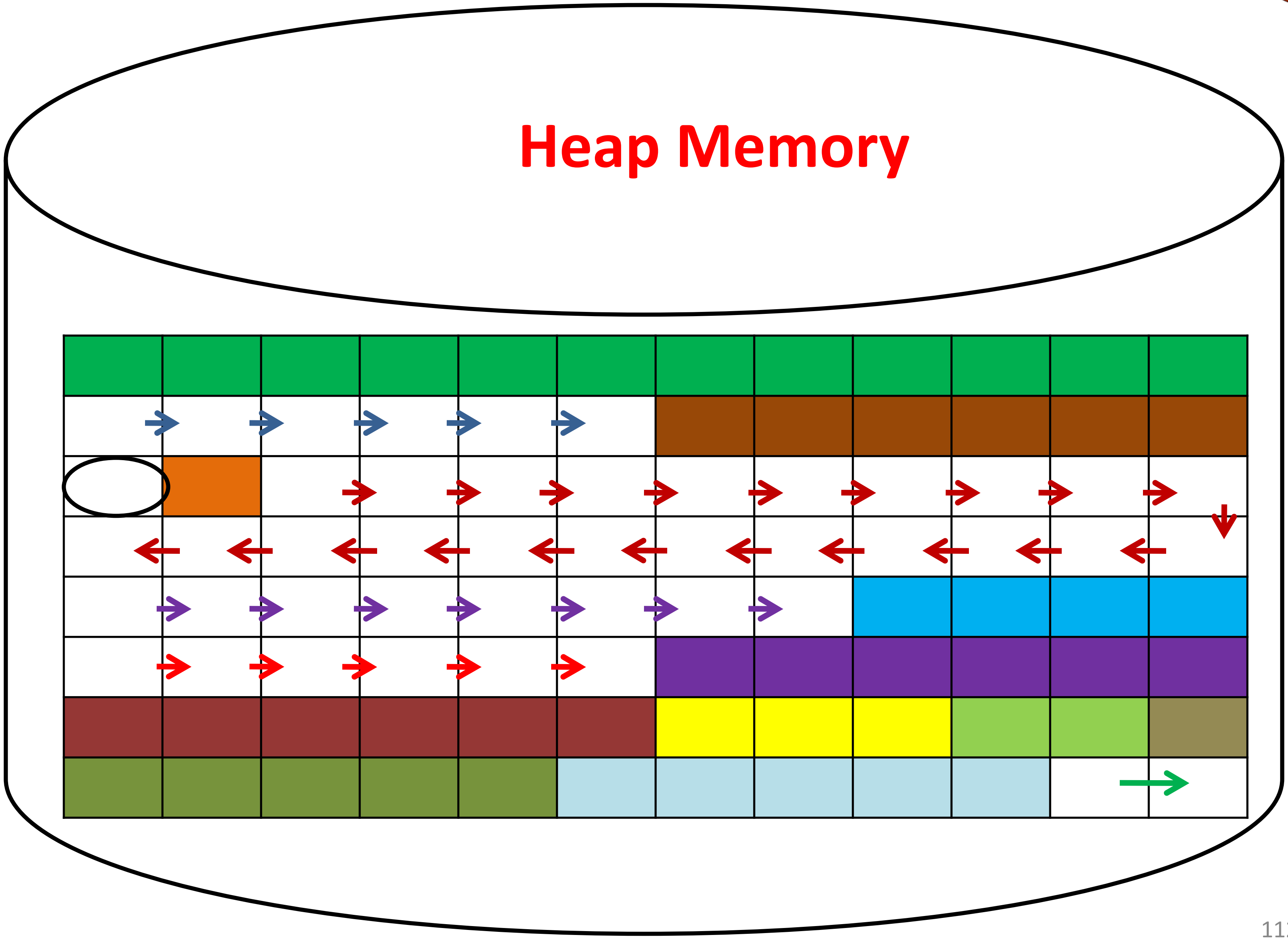
```
for every pointer r do
    mark(r)

void mark(void * ptr) {
    if (ptr != 0)
      if (*ptr.marker is not marked) {
        set *ptr.marker
        mark(*ptr.llink)
        mark(*ptr.rlink)
      }
}
```



Mark-and-Sweep Process

- It was done too infrequently -> Only when a program had used all or nearly all of the heap storage
    - Takes a good deal of time
    - May yield only a small number of cells that can be placed on the list of available space
- Incremental Mark-Sweep -> Garbage collection occurs more frequently long before memory is exhausted
    - More effective in terms of the amount of storage that is reclaimed
    - Time required for each run of the process is obviously shorter, thus reducing the delay in application execution
- Perform the mark-sweep process on parts at different times

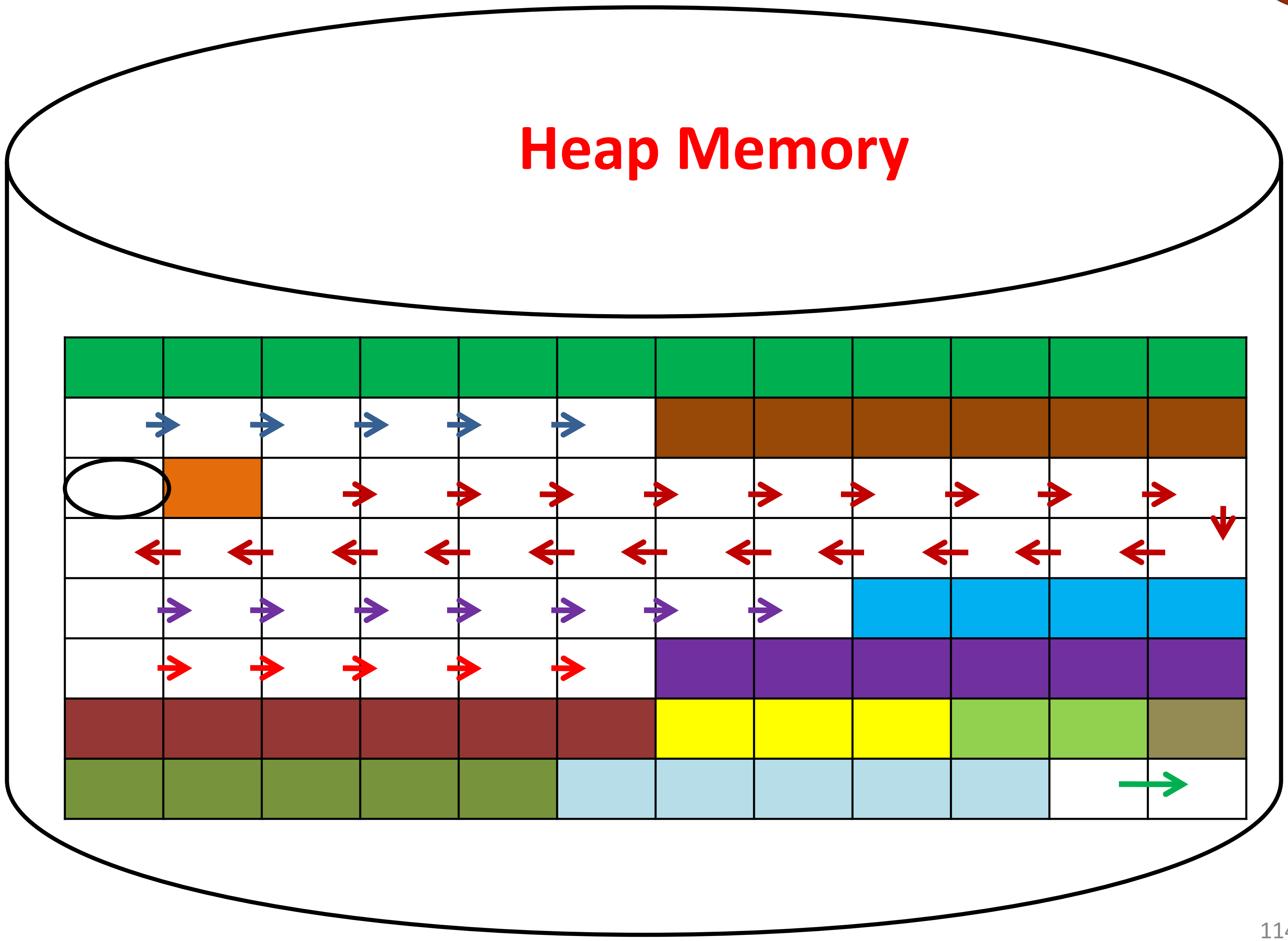
Variable-Size Cells



Variable-Size Cells

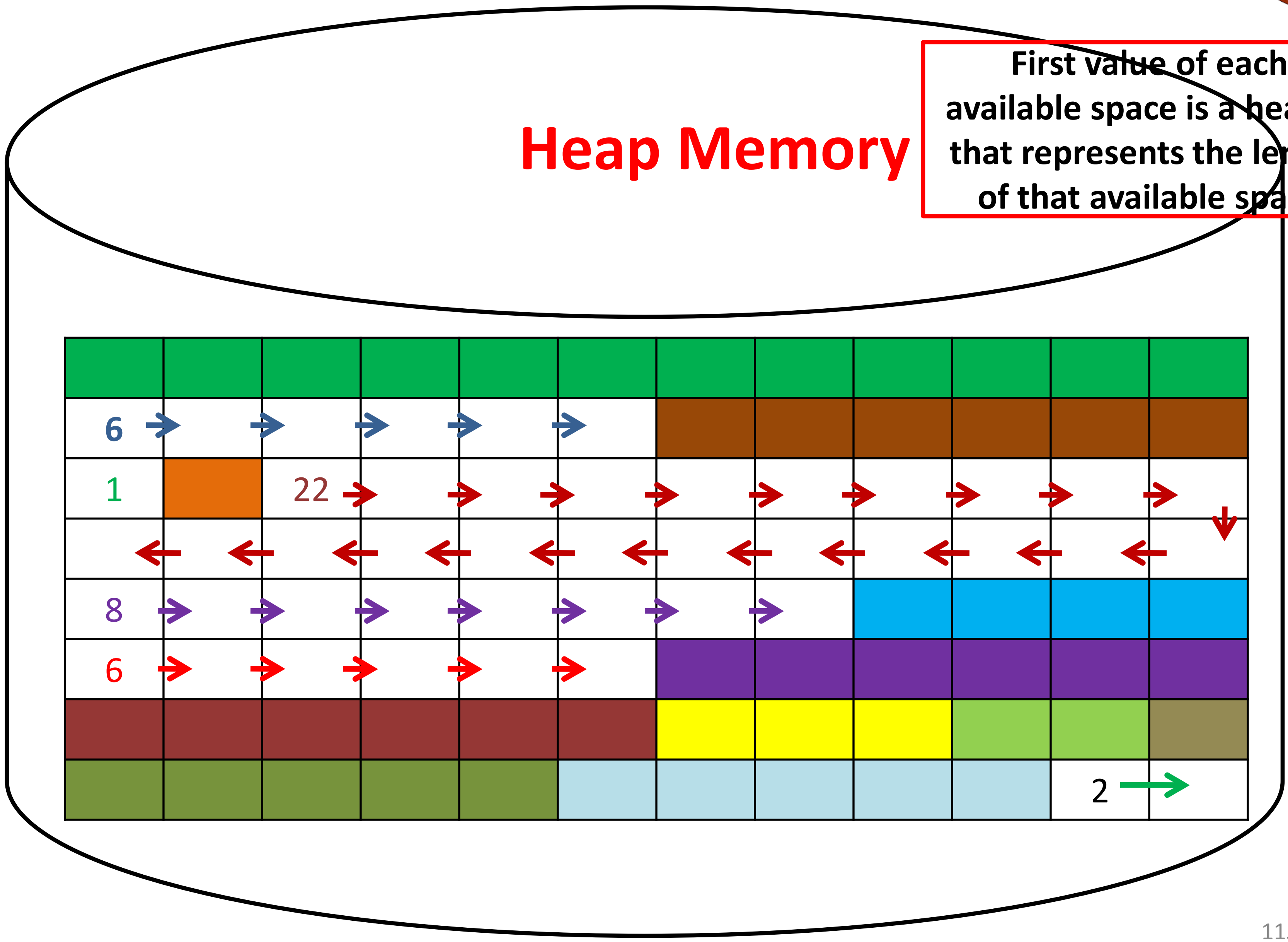
- Managing a heap from which variable-size cells are allocated has all the difficulties of managing one for single-size cells, but also has additional problems
- Additional problems posed by variable-size cell management depend on the method used
- If mark-sweep is used, the following additional problems occur
  - Initial setting of the indicators of all cells in the heap to indicate that they are garbage is difficult
    - Because the cells are different sizes, scanning them is a problem
  - One solution is to require each cell to have the cell size as its first field
    - Then the scanning can be done, although it takes slightly more space and somewhat more time than its counterpart for fixed-size cells

Variable-Sized Cells



Variable-Sized Cells

First value of each available space is a header that represents the length of that available space



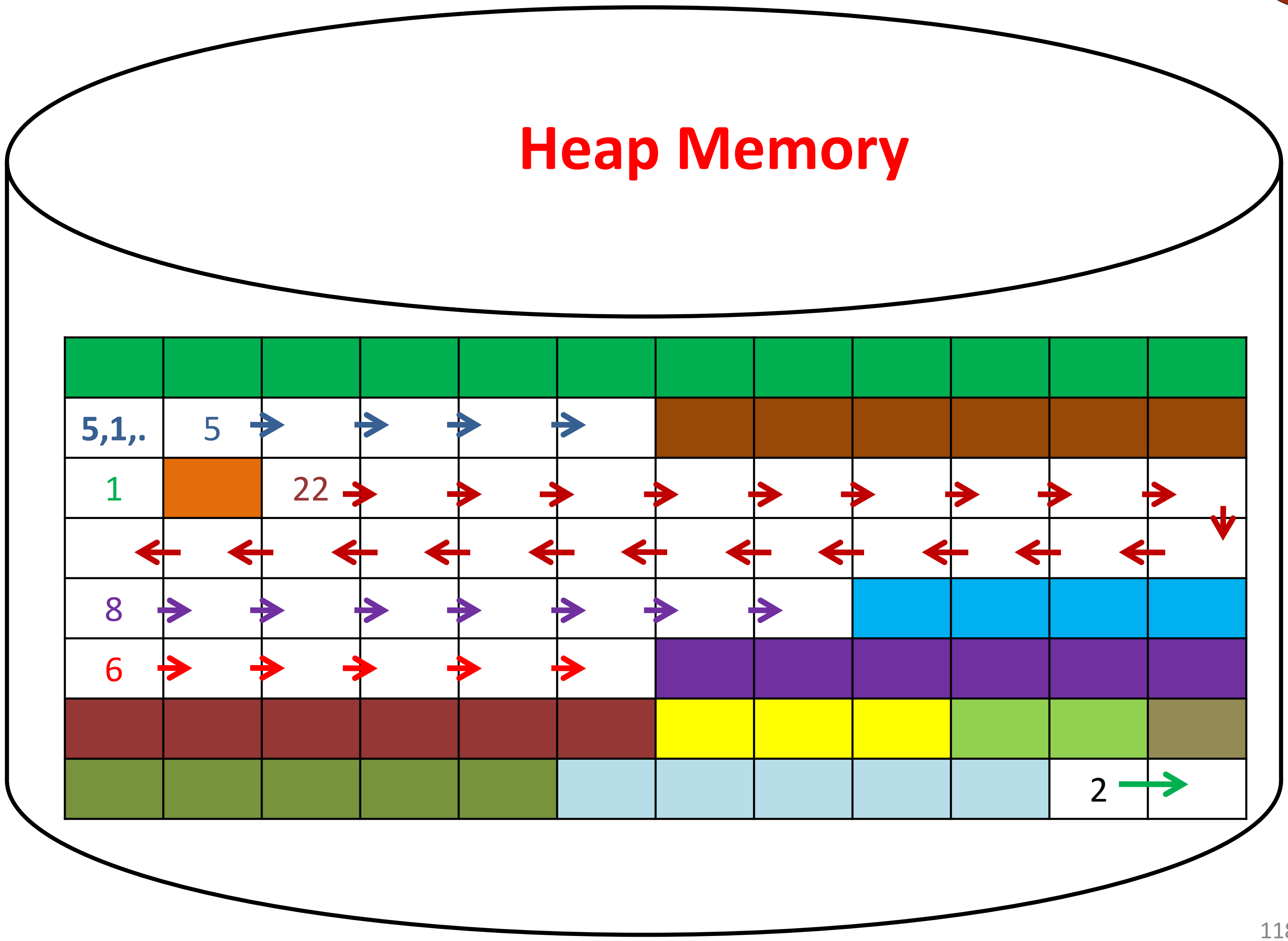
Variable-Size Cells

- Marking process is nontrivial
  - How can a chain be followed from a pointer if there is no predefined location for the pointer in the pointed-to-cell?
  - Cells that do not contain pointers at all are also a problem
    - Adding an internal pointer to each cell, which is maintained in the background by the run-time system, will work
    - However, this background maintenance processing adds both space and execution time overhead to the cost of running the program

Variable-Size Cells

- Maintaining the list of available space is another source of overhead
  - The list can begin with a single cell consisting of all available space
  - Requests for segments simply reduce the size of this block
  - Reclaimed cells are added to the list
- The problem is that before long, the list becomes a long list of various-size segments, or blocks
- This slows allocation because requests cause the list to be searched for sufficiently large blocks
- Eventually, the list may consist of a large number of very small blocks, which are not large enough for most requests
- At this point, adjacent blocks may need to be collapsed into larger blocks
- Alternatives to using the first sufficiently large block on the list can shorten the search but require the list to be ordered by block size
- In either case, maintaining the list is additional overhead
- If reference counters are used, the first two problems are avoided, but the available-space list-maintenance problem remains

Variable-Sized Cells



Miscellanoeus

Internal Fragmentation

- Difference between memory allocated and required space or memory is called Internal Fragmentation

External Fragmentation

- Unused spaces formed between non-contiguous memory fragments are too small to serve a new request is called External fragmentation