COMP 348

Principles Of Programming Languages

Review

PROLOG

Predicates, Relations, Facts, Procedure

```
parent(tom, adam).
parent(tom, helen).
parent(sandra, adam).
parent(sandra, helen).
parent(michael, andrew).
parent(michael, john).
```

Prolog programs consist of collections of statements (called assertions, or clauses).

Statements are grouped into procedures. In the example, we have one procedure named 'parent', made up of several statements.

Each procedure defines a certain relationship between its arguments.

The programmer decides on how to interpret this relationship. Here parent(tom, adam) will be interpreted as "Tom is the parent of Adam."

PROLOG

Predicates, Relations, Facts, Procedure

```
parent(tom, adam).
parent(tom, helen).
parent(sandra, adam).
parent(sandra, helen).
parent(michael, andrew).
parent(michael, john).
```

The collection of statements constitutes a (declarative) database.

We can pose queries on this database.

A query is the codification of a question.

There are only two types of queries:

1. *Is it indeed the case that a given statement is true?* (ground query)

2. *Under what conditions, if any, is a given statement true?* (non-ground query)

PROLOG

Predicates, Relations, Facts, Procedure

```
parent(tom, adam).
parent(tom, helen).
parent(sandra, adam).
parent(sandra, helen).
parent(michael, andrew).
parent(michael, john).
```

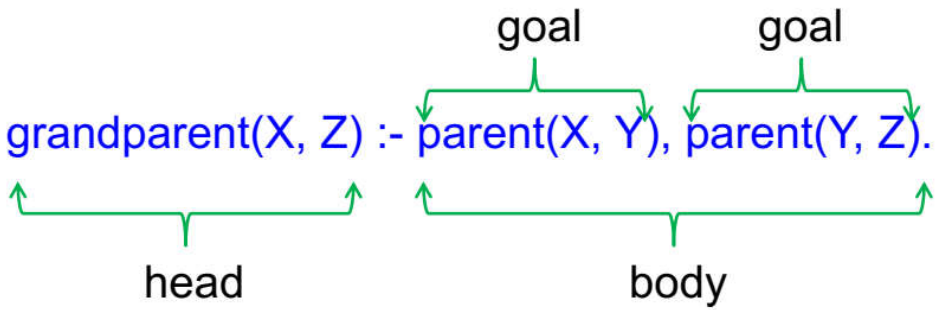
Let p stand for isParentOf relation and let g stand for isGrandParentOf relation.

We can define g in terms of p by the following formula: For persons x, y, z:

$$G = \forall\, x\, \forall\, y\, \forall\, z\, ((p(x,z) \wedge p(z,y)) \rightarrow g(x,y))$$

We can represent the formula with the rule below:

```
grandparent(X, Z) :- parent(X, Y), parent(Y, Z).
```



The goals are related by a conjunction (denoted by the comma symbol).

PROLOG

Predicates, Relations, Facts, Procedure

```
parent(tom, adam).
parent(tom, helen).
parent(sandra, adam).
parent(sandra, helen).
parent(michael, andrew).
parent(michael, john).
parent(judy, roger).
parent(judy, jim).
parent(judy, janis).
parent(mark, roger).
parent(mark, jim).
parent(mark, janis).
parent(janis, daphne).
parent(peter, daphne).
```

Prolog will search its database from top to bottom and

- a) **unify** the query with the head of the rule,

b) **instantiate** X to judy and Z to daphne,

```
grandparent(X, Z) :- parent(X, Y), parent(Y, Z).
```

```
?- grandparent(judy, daphne).
```

PROLOG

Unification

term1 and **term2** unify iff:

1. If term1 and term2 are **constants**, then they unify only if they are the same atom, or the same number.

2. If term1 is a **variable** and term2 is any type of term, then they unify, and term1 is instantiated to term2. (And vice versa.) (*If they are both variables, they're both instantiated to each other, and we say that they share values.*)

3. If term1 and term2 are **complex terms**, they unify only if:

a. They have the same **functor** and **arity**.

b. All of their corresponding arguments unify. **Recursion!**

c. Same variable is *not given two different unifications/values*).

foo(a,b) = foo(X,Y). 2*3+4 = X+Y. [a,b,c] = [a|[b|[c|[]]]].

foo(a,b) = foo(X,X).

PROLOG

Structures

person(john, cohen, date(17,may,1990), unemployed)
dateofbirth(person(_, _, Date, _), Date).

person(john, cohen, date(17,may,1990), unemployed)
dateofbirth(P, date(_,_,Y)), Y<1963

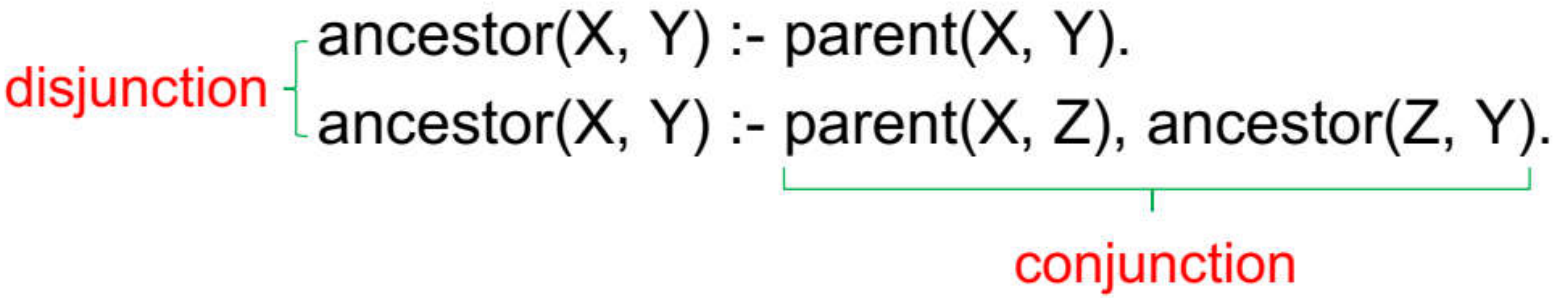
PROLOG

Evaluation

Declarative and Procedural

ancestor(X, Y) iff parent (X, Z) and ancestor (Z, Y)

Solve parent first for Z and then ancestor for Y



PROLOG

Lists

The symbol | in [H|T] represents a list whose head is H and whose tail is T.

We can represent the above example as [john | [eve, paul]]

Since [eve, paul] is also a list with head eve and tail [paul], we can write the above list as [john | [eve | [paul]]]

Any one-element list can be written as that element joined to the empty list. Thus, [paul] is the same as [paul | []]

We can now write the full list as [john | [eve | [paul | [ ]]]]

[a, b, c, d, e] = [a|[b, c, d, e]] = [a | [b | [c, d, e]]] = [a | [b | [c| [d, e]]]]

PROLOG

Lists

The symbol | in [H|T] represents a list whose head is H and whose tail is T.

We can represent the above example as [john | [eve, paul]]

Since [eve, paul] is also a list with head eve and tail [paul], we can write the above list as [john | [eve | [paul]]]

Any one-element list can be written as that element joined to the empty list. Thus, [paul] is the same as [paul | []]

We can now write the full list as [john | [eve | [paul | [ ]]]]

[a, b, c, d, e] = [a|[b, c, d, e]] = [a | [b | [c, d, e]]] = [a | [b | [c| [d, e]]]]

first(H,[H|_]):-true. third(X,[_,_,X|_]). third(X,[_|[_|[X|]]).

PROLOG

Recursion

```
member(X,[X|_]).
member(X,[_|T]) :- member(X, T).

last(L,[L]).
last(L,[_|T]):-last(L,T).

secondLast(L, [L,_]).
secondLast(L, [_|T]):-secondLast(L, T).

size([], 0).
size([_|T], S) :- size(T, S1), S is S1+1.

pow(_, 0, 1).
pow(1, _, 1).
pow(X, Y, Z) :- N is Y-1, pow(X, N, W), Z is W*X.
```

PROLOG

Cut operator !

The cut, in Prolog, is written as !, a goal, which always succeeds, but cannot be backtracked. It is used to prevent unwanted backtracking.

```
teaches(dr_fred, history).        studies(alice, english).
teaches(dr_fred, english).        studies(angus, english).
teaches(dr_fred, drama).          studies(amelia, drama).
teaches(dr_fiona, physics).       studies(alex, physics).
```

?- teaches(dr_fred, Course), studies(Student, Course), !.
Course = english
Student = alice ;
false.

?- teaches(dr_fred, Course), !, studies(Student, Course).
False

?- !, teaches(dr_fred, Course), studies(Student, Course).

LISP

append, cons, list

```
(cons ‘a, ‘(b, c))          ; return (A B C)
(cons 'a '(b) 'c)           ; Too many arguments
(cons 'a '(b))              ; (A. B)
(cons 'a '(b))              ; (A B)
```

```
(list 1 2 ’a 3)                  ; Returns (1 2 A 3).
(list 1 ’(2 3) 4)                ; Returns (1 (2 3) 4).
(list ’(+ 2 1) (+ 2 1))          ; Returns ((+ 2 1) 3).
(list 1 2 3 (list ’a ’b 4) 5) ; Returns (1 2 3 (a b 4) 5).

(append ’(1 2) ’(3 4))                ; Returns (1 2 3 4).
(append ’(1 2 3) ’() ’(a) ’(5 6)) ; Returns (1 2 3 a 5 6).
```

```
> (append (list 1) ’(4 5 6))
(1 4 5 6)
```

LISP

Variable Binding

```
(let ((x 2) (y 3))
    (+ x y))

; Returns 5.
```

```
(let ((a 1))
   (let ((a 2))
      (let ((a 3))
         ...)))
```

```
(let ((x 1))                    ; x is 1.
     (let ((x (+ x 1)))         ; x is 2.
     (+ x x)))                  ; Returns 4.
```

```
(let* ((x 10)
       (y (* 2 x))) ; Not legal for let.
       (* x y))

; Returns 200.
```

LISP

apply, mapcar, funcall

`mapcar` takes as its arguments a function and one or more lists and applies the function to the elements of the list(s) in order.

```
; Multiplication applies to successive pairs.
> (mapcar #'* '(2 3) '(10 10))
(20 30)
```

```
> (funcall #'+ 1 3 4)  ; Equivalent to (+ 1 3 4).
8
```

`apply` works like `funcall`, but requires that the last argument is a list.

```
> (apply #'+ 3 4 '(1 3 4))
15
```

LISP

Anonymous Functions

An anonymous function can be applied in the same way that a named function can, e.g.

```
> ((lambda (x) (* x x)) 3)
9
```

```
>  (mapcar (lambda (n) (* n 2)) '(2 3 5 7))
(4 6 10 14)
```

LISP

Structure Change

```
> x
(A B C D E)
> (setf y '(a b c d e))
(A B C D E)
> (eql x y)
NIL
> (equal x y)
T
> (setf z x)
(A B C D E)
> (eql x z)
T
> (equal x z)
T
> (eql y z)
NIL
> (equal y z)
T
```

```
> (setf x '(a b c))
(A B C)
> (car x)
A
> (cdr x)
(B C)
> (cdr (cdr (cdr x)))
NIL
> (setf x (append x '(d e)))
(A B C D E)
```

LISP

Recursion

```
(defun append2 (lst1 lst2)
  (if (null lst1)
    lst2
    (cons (car lst1) (append2 (cdr lst1) lst2))))

(defun last2 (lst)
  (cond ((null lst) nil)
        ((null (cdr lst)) (car lst))
        (t (last2 (cdr lst)))))

(defun reverse2 (lst)
  (cond ((null lst) '())
    (t (append (reverse2 (cdr lst)) (list (car lst))))))
```

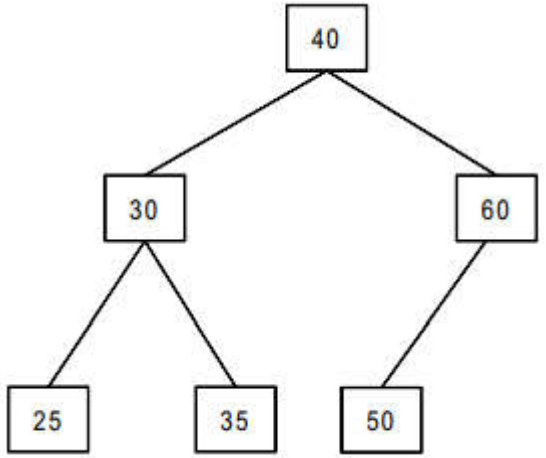
LISP

Binary Trees

We can represent the entire tree as one single list:

```
'(40                    ; Root.
    (30                 ; Root of left subtree.
       (25 () ())
       (35 () ())
    )
    (60                 ; Root of right subtree.
       (50 () ())
       ()
    )
 )
```

or `'(40 (30 (25 () ())(35 () ()))(60 (50 () ())()))`



RUBY

Chaining and Parallel Assignments

```
a = b = 1 + 2 + 3
puts a #=> 6
puts b #=> 6
a = (b = 1 + 2) + 3
puts a #=> 6
puts b #=> 3
```

```
a = 1
b = 2
a, b = b, a
puts a #=> 2
puts b #=> 1
x = 0
a, b, c = x, (x += 1), (x += 1)
puts a #=> 0
puts b #=> 1
puts c #=> 2
puts x #=> 2
```

RUBY

Arrays

```
myarray = [ 1, 2, 3, 4, 5, 6 ]

     index  0  1  2  3  4  5

puts myarray[0]       #=> 1

                      # [i...j] from index i to j
                      # excluding j
puts myarray[1...3]   # Exclusive range. => 2 3.

                      # [i..j] from index i to j
                      # including j
puts myarray[1..3]    # Inclusive range. => 2 3 4.

puts myarray[1,3]     # Range between 1st up to 3rd
                      # consecutive, inclusive.
                      #=> 2 3 4.
```

```
a = [ "pi", 3.14, "prime", 17 ]
puts a.class      #=> Array
puts a.length     #=> 4
puts a[0]         #=> pi
puts a[-1]        #=> 17
puts a[1]         #=> 3.14
puts a[2]         #=> prime
puts a[3]         #=> 17
puts a[4]         #=> nil
b = Array.new
puts b.class      #=> Array
puts b.length     #=> 0
b[0] = "a"
b[1] = "new"
b[2] = "array"
puts b            #=> a new array
```

RUBY

Associative Arrays

```
biblio["nietzsche97"] = "Beyond good and evil"
puts biblio["nietzsche97"] #=> Beyond good and evil
puts biblio.length         #=> 6
```

```
biblio.each_pair do |key, value|
  puts "#{key} : #{value}"
end
```

```
biblio.each do |key, value|
  puts "#{key} : #{value}"
end
```

```
biblio.each {|key, value| puts key + " : " + value}
```

```
biblio.each_key {|key| puts key}
```

```
biblio.delete_if {|key, value| key == "kafka95"}
puts biblio.length         #=> 5
```

RUBY

Classes

```
class Coordinate
  attr_accessor :x, :y
  @@total = 0
  def initialize (x, y)
    @@total += 1
    @x = x
    @y = y
  end
  def to_s
     return "(#@x, #@y)"
  end
  def Coordinate.total
    return "Number of coordinates:  #@@total"
  end
end
```

```
local_variable
CONSTANT_NAME / ConstantName / Constant_Name
:symbol_name
@instance_variable
@@class_variable
$global_variable
ClassName
method_name
ModuleName
```

```
attr_writer :x
attr_reader :y

def x= (value)
    @x = value
end

def y
    @y
end
```

RUBY

Classes

require to include external file, similar to import in Java

```
require "CoordinateV2.rb"
class XYZCoordinate < Coordinate
```

include to include mixins/modules

```
# Module defined in util.rb
module Utility
    def Utilty.logMe(details)
        # …
    end
end

require 'util.rb'  #.rb is optional
include Utility
Utility.logMe("This is a test.")
```

RUBY

Object Extensions

Ruby allows us to extend specific instances with new behavior. Consider the example below:

```
def p1.whatIam
  return "The origin on the 3D system."
end

puts p1.whatIam #=> The origin on the 3D system.
puts p2.whatIam #=> Will cause an error.
```

RUBY

Control flow

```
if boolean-expression-1 [then]
   if-body
[else boolean-expression-2 [then]
   else-body]
end
```

```
if boolean-expression-1 [then]
   if-body
elseif boolean-expression-2 [then]
   elseif-body
   ...
[else boolean-expression-n [then]
   else body]
end
```

The term *unless* works as a negated *if*.

```
unless boolean-expression [then]
    unless-body
[else
    else-body]
end
```

$boolean - expression\ ?\ expression_1\ :\ expression_2$

RUBY

Control flow

```
case target
    when comparison [, comparison ] ... [ then ]
        body
    when comparison [, comparison ] ... [ then ]
        body
    ...
      [ else
          body ]
end
```

```
number = 11
case number
  when 1, 3, 5, 7, 9
    puts "Odd."
  when 0, 2, 4, 6, 8, 10
    puts "Even."
  else
    puts "Number is out of range."
end
```

RUBY

Repetitions

```
while boolean-expression [ do ]
   body
end
```

```
until boolean-expression [ do ]
   body
end
```

```
a = [ "3.14", "number", "pi" ]
a.each { |el| print el + " " }
```

```
3.times { |count| puts count}

1.upto(10) { |count| puts count }

10.downto(1) { |count| puts count }

0.step(10,2) { |count| puts count }

for element in ['a', 'b', 'c']
  puts element
end
```

```
loop do
   body
   next if boolean-expression  # skip iteration
   break if boolean-expression # exit loop
   redo if boolean-expression  # do it again
end
```

RUBY

Regular Expressions

- A regular expression is a way of specifying a pattern of characters to be matched in a string.
- In Ruby this is done with `/pattern/`.
- In Ruby, regular expressions are objects and can thus be manipulated as such. Some common pattern descriptions are shown below:

Pattern	Description
`/Lisp|Lava/`	Matches a string containing *Lisp*, or *Lava*.
`/L(isp|ava)/`	As above.
`/ab+c/`	Matches a string containing an *a*, followed by one or more *b*s, followed by a *c*.
`/ab*c/`	matches a string containing an *a*, followed by zero or more *b*s, followed by a *c*.
`.`	Matches any character.
`/[Colloqui[um|a]/`	Matches *Colloquium*, or *Colloquia*.

RUBY

Introspection

- *Introspection* is the ability of a computational system to consult (but not modify) its own structure.
- In Ruby, we can obtain the following type of knowledge about a program:
  - What objects it contains.
  - The contents and behaviors of objects.
  - The current class hierarchy.

```
puts p1.id                          #=> 21113660
puts p1.class                       #=> Coordinate
puts p2.class                       #=> XYZCoordinate
puts p2.instance_variables          #=> @y @z @x
puts p2.kind_of? Coordinate         #=> true
puts p2.kind_of? XYZCoordinate      #=> true
puts p1.kind_of? XYZCoordinate      #=> false
puts p2.instance_of? Coordinate     #=> false
puts p2.instance_of? XYZCoordinate #=> true
```

C

Declaration and Definition

```
#include<stdio.h>
long factorial(int);
int main() {
  int n;
  long f;
  printf("Enter an non-negative integer: ");
  scanf("%d", &n);
  if (n < 0)
    printf("Negative integers are not allowed.\n");
  else {
    f = factorial(n);
    printf("%d! = %ld\n", n, f);
  }
  return 0; }
long factorial(int n) {
  if (n == 0)
    return 1;
  else
    return(n * factorial(n-1)); }
```

Declaration

Definition

C

Global and Local variables

```
#include<stdio.h>
int a = 3;
int func() {
  int a = 5;
  return a;
}
int main() {
  printf("From main: %d\n", a);
  printf("From func: %d\n", func());
  printf("From main: %d\n", a);
}
```

- The output is as follows:

```
From main: 3
From func: 5
From main: 3
```

C

Global and Local variables

```
#include<stdio.h>
int a = 3;
int func() {
  int a = 5;
  return a;
}
int main() {
  printf("From main: %d\n", a);
  printf("From func: %d\n", func());
  printf("From main: %d\n", a);
}
```

- extern: Variable/function is defined outside of current file.
- (blank): Variable/function is defined in current file and visible outside.
- static: Variable/function is visible only in current file.

- The output is as follows:

```
From main: 3
From func: 5
From main: 3
```

The C Standard Library

- An *application programming interface* (API) is a protocol that constitutes the interface of software components.
- In the C language this is a collection of functions grouped together according to their domain.
- We can access this API (called the *C standard library*) by adding the `#include` directive at the top of our program file.
- Perhaps the most common is the group of functions that support input-output and are accessed by `<stdio.h>`.
- math.h: Defines common mathematical functions.
- stdio.h: Defines core input and output functions.
- stdlib.h: Defines numeric conversion functions, pseudo-random number generation functions, memory allocation, process control functions.
- string.h: Defines string manipulation functions.

C

Primitive Data Types

IDENTIFIER	TYPE	RANGE
int	integer	$-32,768$ to $32,767$
float	real	1.2×10^{-38} to 3.4×10^{38}
double	real	2.2×10^{-308} to 1.8×10^{308}
char	character	ASCII

$$short\ int \leq int \leq long\ int$$

$$float \leq double \leq long\ double$$

OPTIONAL SPECIFIER	RANGE
short int	$-32,768$ to $32,767$
unsigned short int	0 to $65,535$
unsigned int	0 to $4,294,967,295$
long int	$-2,147,483,648$ to $2,147,483,647$

Pointers

- A *pointer* is a type that references (*"points to"*) another value by storing that other value's address.
- A *pointer variable* (also called an *address variable*) is declared by putting an asterisk * in front of its name, as in the following statement that declares a pointer to an integer.

```
int *ptr;
```

- There are two operators that are used with pointers:
  - * The "dereference" operator: Given a pointer, obtain the value of the object referenced (pointed at).
  - & The "address of" operator: Given an object, use & to point to it. The & operator returns the address of the object pointed to.

```
#include <stdio.h>
int main() {
  int a = 42;
  int *p;
  p = &a;
  printf("p: %d\n", *p);
  return 0;
}
```

```
#include<stdio.h>
int main() {
  int my_var = 13;
  int *ptr = &my_var;
  *ptr = 17;
  printf("my_var: %d\n", my_var);
}
```

C

Pointers and Arrays

```
int arr[5];
int *ptr;
```

In the following statement, we assign the first element of the array as the value of the pointer:

```
ptr = &arr[0];
```

Pointer arithmetic makes `*(ptr + 1)` the same as `arr[1]`.

```
#include <stdio.h>
int main() {
  int arr[5] = {1, 3, 5, 7, 11};
  int *ptr;
  ptr = &arr[0];
  printf("arr[0]: %d, arr[1]: %d, arr[2]: %d\n",
         *ptr, *(ptr + 1), *(ptr + 2));
  return 0;
}
```

`*(ptr + 1)` is not the same as `*(ptr) + 1`.

Function Pointers

- Pointers to variables and arrays are examples where a pointer refers to data values.
- A pointer can also refer to a function, since functions have addresses.
- We refer to these as *function pointers*. Consider the following (rather cryptic) declaration:

  ```
  long (*ptr)(int);
  ```

- This declares a function pointer; It points to a function that takes an integer argument and returning a long integer.

```
#include <stdio.h>
long factorial(int);
int main() {
  int n;
  long f;
  long (*ptr)(int);
  ptr = &factorial;
```

```
long factorial(int n) {
  if (n == 0)
    return 1;
  else
    return(n * factorial(n-1)); }
```

C

Structures

A *record*, or *structure*, is a collection of elements, *fields* (or *members*), which can possibly of different types.

The syntax of declaring a structure in C is

```
struct <name> {
  field declarations
};
```

To create a structure to represent a coordinate on the Cartesian plane, we can say:

```
struct coordinate {
  float x;
  float y;
};
```

To create a coordinate variable we can now say

```
struct coordinate p;
```

Structures

Members of the coordinate type can be initialized during declaration either inline as in

```
coordinate p1 = {0, 0};
```

or by designated initializers as in

```
coordinate p2 = {.x = 1, .y = 3};
```

Subsequently members of a record can be assigned values as in

```
p3.x = 2;
p3.y = 7;
```

or by assigning the value of one record to another, as in

```
p4 = p3;
```

that copies the member values from p3 into p4.

C

Structures and Pointers

A pointer can be deployed to point to a record as in

```
coordinate p = {0, 0};
coordinate *ptr = &p;
```

The pointer can subsequently be dereferenced using the * operator as in

```
(*ptr).x = 3;
```

An alternative binary operator exists (->): The left operand dereferences the pointer, where the right operand accesses the value of a member of the record:

```
ptr->y = 3;
```

Structures and Arrays

`line[]` is an array of type `coordinate`, itself defined as a record.

The elements of the array are initialized at the time of declaration.

We use the dot (.) operator to access fields of individual records: `line[0].x` accesses the `x` field of the first element (record) of `line`.

```
#include<stdio.h>
typedef struct {
  float x;
  float y;
} coordinate;
```

```
int main() {
  coordinate line[2] = {
    {0, 0},
    {11, 19}
  };
  printf("Line points: (%.0f, %.0f), and (%.0f, %.0f).\n",
          line[0].x, line[0].y, line[1].x, line[1].y );
}
```

C

Memory Management

```
#include<stdio.h>
#include <stdlib.h>
int main() {
  int *array = malloc(3 * sizeof(int));
  if (array == NULL) {
    printf("ERROR: Out of memory.\n");
    return 1;
  }
  *array = 1;
  *(array + 1) = 3;
  *(array + 2) = 5;
  printf("%d\n", *array);
  printf("%d\n", *(array + 1));
  printf("%d\n", *(array + 2));
  free(array);
  return 0;
}
```

`int *array = malloc(3 * sizeof(int));`

we request the allocation of enough memory for an array of three elements of type `int`.

As a result, to indicate success we now have to verify that our `array` pointer is not `NULL`.

`if (array == NULL) {...}`

Once we no longer need the array, we have to release the allocated memory back to the system.

C

Memory Management

malloc: Allocates the specified number of bytes.

realloc: Increases or decreases the size of the specified block of memory.

calloc: Allocates the specified number of bytes and initializes them to zero.

free: Releases the specified block of memory back to the system.

```
char *str;
str = (char *) malloc(9);
strcpy(str, "COMP348");

str = (char *) realloc(str, 25);
strcat(str, "_FinalExam");
printf("String = %s,  Address = %u\n", str, str);
free(str);
```

C

Linked List

```
int main() {
  struct node *head = NULL;
  struct node *new;
  head = malloc(sizeof(struct node));
  if (head == NULL) {...}
  head->data = 5;
  head->next = NULL;
  new = malloc(sizeof(struct node));
  if (new == NULL) {...}
  new->data = 11;
  new->next = head;
  head = new;
  printf("%d ", head->data);
  printf("%d ", (head->next)->data);
  return 0;
}
```

```
#include<stdio.h>
#include <stdlib.h>
struct node {
  int data;
  struct node *next;
};
```

Best of Luck !!!