CS1010E Lecture 8

Composite Data Types: Structures and Arrays

Henry Chia (hchia@comp.nus.edu.sg)

Semester 1 2016 / 2017

Lecture Outline

□ Structures

- Structure definition and declaration
- Assigning structures
- Accessing members of a structure
- Passing structures by value to a function
- Returning a structure from a function

□ Arrays (one-dimensional)

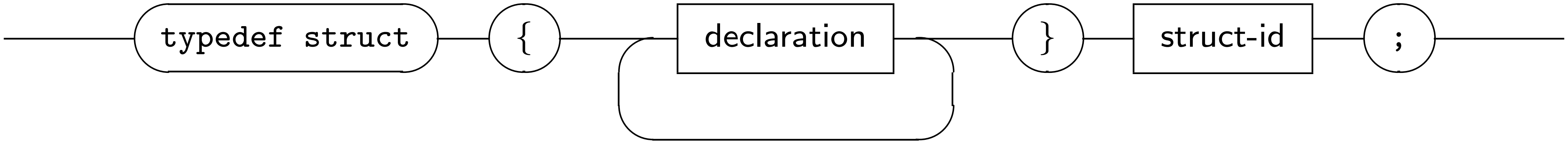
- Declaration and initialization
- Array element access
- Array as function argument

Structures

- □ Declaring primitive variables of `int`, `double` or `bool` allow us to work with individual values
- □ To deal with practical problems, each variable or value may constitute a set of information/data — *record*
  - – A point comprises two floating-point values
  - – A fraction comprises two integers
  - – A bank account is associated with an integer account number, and a floating-point balance.
- □ A **structure** defines a set of *heterogeneous* data for a record, i.e. the individual parts of the data do not have to be of the same type

Structure Definition

struct-def



- □ Examples of a structure definitions:

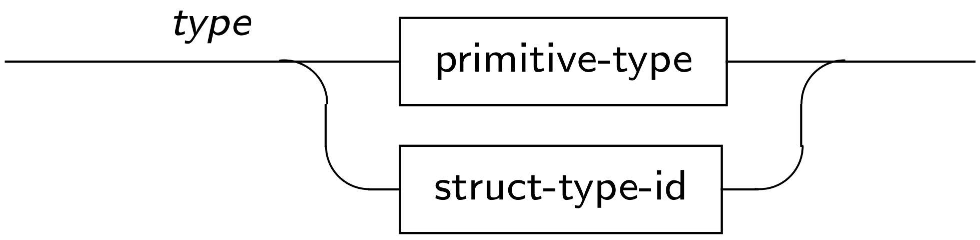
```
typedef struct {
   double x, y;
} Point;
```

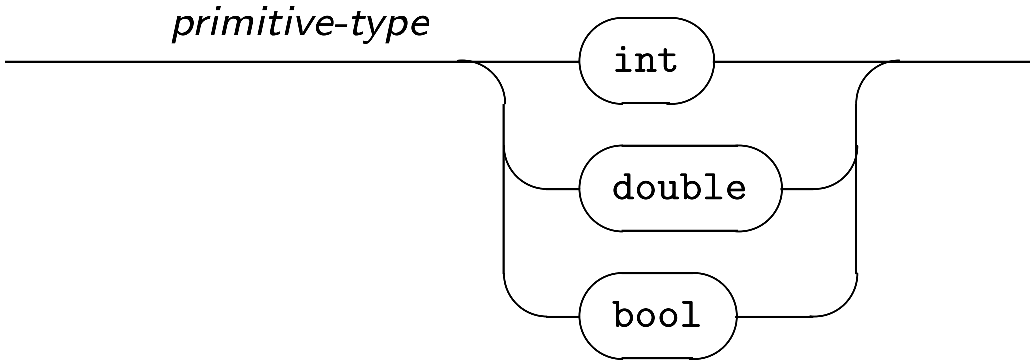
```
typedef struct {
   int num, den;
} Fraction;
```

```
typedef struct {
   int accountNum;
   double balance;
} BankAccount;
```

- □ Structures are aggregate data types as multiple data values are collected into a single data type
- □ Capitalize the initial letter of a structure identifier

Structure Definition





- □ Defining a structure is to define the blueprint for a new *type*; memory is not allocated for structure definitions
- □ Structures are defined before any type declarations, i.e. typically above function prototypes
- □ Unique identifiers within a structure define individual data values, called **data members**

Declaration with Structures

- Declaration of structure-typed variables proceed after the structure definition
- Using the Point structure definition

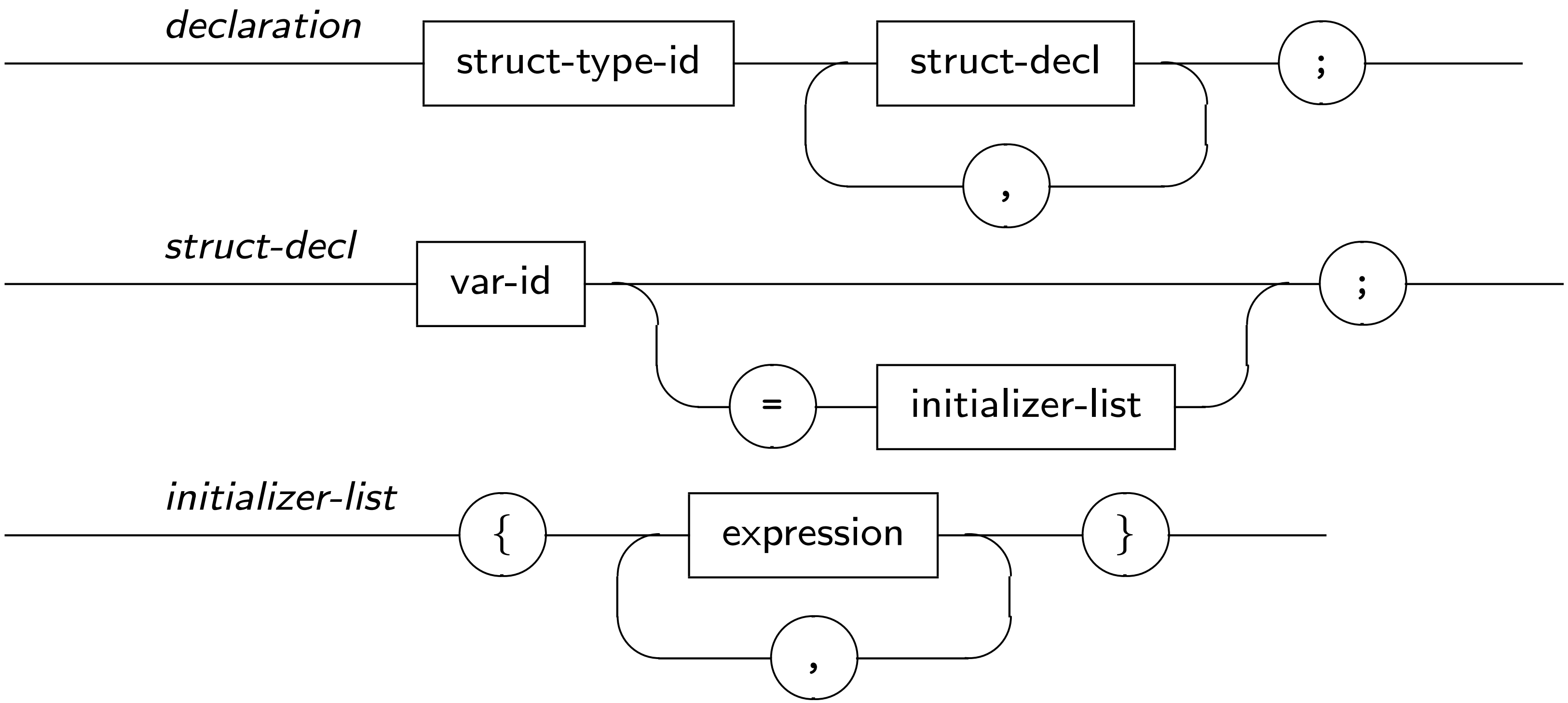
  ```
  typedef struct {
    double x, y;
  } Point;
  ```

  we may now proceed to declare

  ```
  Point p, q;
  ```

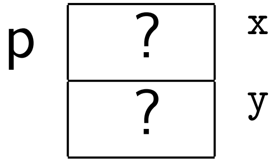
- In the above, each variable p and q contains two data values — the x and y values associated with the point
- A structure variable is modeled as a box, just like a primitive int variable, but with richer content

Declaration with Structures



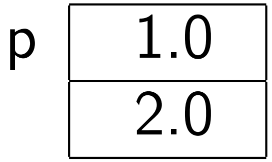
- □ Without initialization:

```
Point p;
```



- □ Using an initializer list

```
Point p = {1.0, 2.0};
```



- □ To zero a structure, just zero the first data member

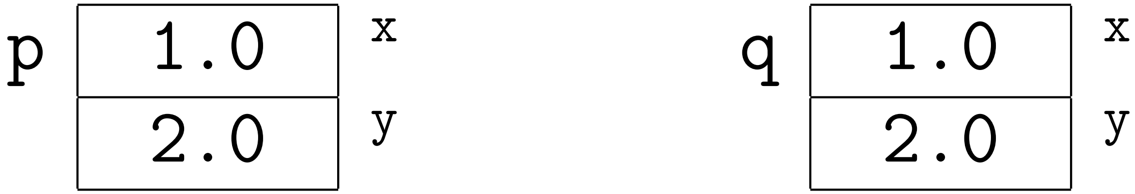
```
Point p = {0.0};
```

Assignment with Structures

- □ Like primitives, a structure variable can be assigned to another structure variable of the same type

```
Point p = {1.0, 2.0}, q;
```

```
q = p;
```



- □ Explicit structure values cannot be assigned to a structure directly using initializer lists

```
q = {1.0, 2.0}; /* illegal assignment!! */
```

Member Operator

- □ A data member is referenced using the structure variable name followed by the **structure member operator** (`.`) and a data member name
- □ As such, structure variables can by assigned by “assigning” individual data members

```
q.x = 1.0;
q.y = 2.0;
```

- □ Arithmetic/relational/logical operations **cannot** be applied on entire structure variables; it does not make sense
- □ These operations should be applied on specific members, e.g. condition to compare two points `p` and `q` for equality

```
fabs(p.x - q.x) < EPS && fabs(p.y - q.y) < EPS
```

Member Operator

- Input values into the data members via `scanf`

  ```
  scanf("%lf%lf", &(q.x), &(q.y))
  ```

- Output values of data members via `printf`

  ```
  printf("(%f, %f)\n", q.x, q.y);
  ```

- By applying the member operator, the specific member can be accessed and used in statements, expressions, function arguments, function return values, etc. without regards to it being part of a structure

Structure as Function Input/Output

```c
#include <stdio.h>
#include <math.h>

typedef struct {
   double x, y;
} Point;

double computeDist(Point p, Point q);
Point midPoint(Point p, Point q);

int main(void) {
   Point p, q, m;

   printf("Enter x and y of first point: ");
   scanf("%lf %lf", &(p.x), &(p.y));
   printf("Enter x and y of second point: ");
   scanf("%lf %lf", &(q.x), &(q.y));

   printf("Distance between two points is %lf.\n", computeDist(p, q));
   m = midPoint(p, q);
   printf("Midpoint is (%lf,%lf)\n", m.x, m.y);

   return 0;
}
```

Structure as Function Input/Output

- When a structure variable is used as a function argument, the entire content of the structure is *passed by value*

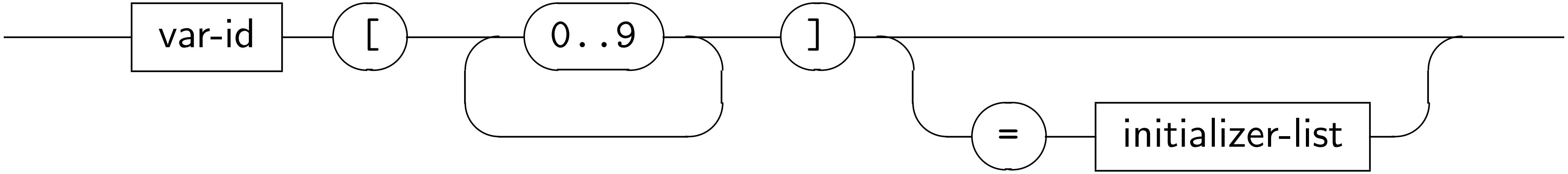
```
double computeDist(Point p, Point q) {
   double dx = p.x - q.x;
   double dy = p.y - q.y;
   return sqrt((dx * dx) + (dy * dy));
}
```

- A function can be defined to return a structure value; the entire structure content is returned

```
Point midPoint(Point p, Point q) {
   Point m;

   m.x = (p.x + q.x) / 2.0;
   m.y = (p.y + q.y) / 2.0;
   return m;
}
```

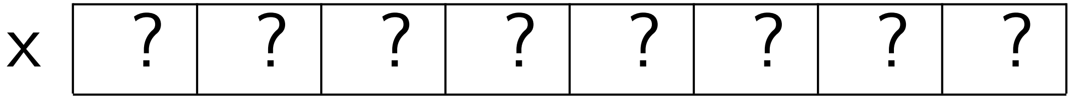
Array Declaration

array-decl



- Use arrays to work with a set of data values of the same type
- Example:

```
double x[8];
```



- An array is a sequential group of memory locations that represent an *ordered* collection of *homogeneous* data, i.e. individual data must be of the same type
  - pre-specified number of elements that must be as large as, or larger than, the maximum number of values stored

Array Initialization

```
double x[8] = {16.0, 12.0, 6.0, 8.0, 2.5, 12.0, 14.0, -54.5};
```

x	16.0	12.0	6.0	8.0	2.5	12.0	14.0	-54.5

□ An array can be initialized using an initializer list:

- specifies the initial values
- used during declaration only
- may be shorter than the array size
- below declarations are equivalent
  - ▷ `double x[8] = {16.0, 0, 0, 0, 0, 0, 0, 0};`
  - ▷ `double x[8] = {16.0};`

□ To *zero* the array, just zero the first element

```
double x[8] = {0.0};
```

Array Subscripts

- Array elements are accessed individually using subscripts
- Subscripts start with 0 and increment by 1
- Example:

	x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
x	16.0	12.0	6.0	8.0	2.5	12.0	14.0	-54.5

  - First value in array `x` is `x[0]`; last value is `x[7]`.

- It is a common mistake to specify a subscript that is outside the bounds of the array:
  - may produce segmentation-fault/bus-error at runtime
  - may cause logic error when a memory location reserved for another variable is modified

Accessing Array Elements

- Example: fill array sq with squared values $0^2, 1^2, 2^2, \ldots, 10^2$

```c
#include <stdio.h>
#define SIZE 11

int main(void) {
   int i, sq[SIZE];

   for (i = 0; i < SIZE; i++) {
      sq[i] = i * i;
      printf("%d ", sq[i]);
   }
   printf("\n");
   return 0;
}
```

- Looping condition must ensure a final loop with i as 10, and not 11, since the array elements are sq[0] through sq[10]

Array as Function Argument

```c
#include <stdio.h>

#define SIZE 11

void printArr(int array[], int n);

int main(void) {
   int i, sq[SIZE];

   for (i = 0; i < SIZE; i++) {
      sq[i] = i * i;
   }
   printArr(sq, SIZE);
   return 0;
}
```

```c
/*
   printArr prints first n
   elements of the array.

   Precondition: n >= 0
*/
void printArr(int array[], int n) {
   int i;

   for (i = 0; i < n; i++) {
      printf("%d ", array[i]);
   }
   printf("\n");
   return;
}
```

□ Passing an array to a function requires two parameters:

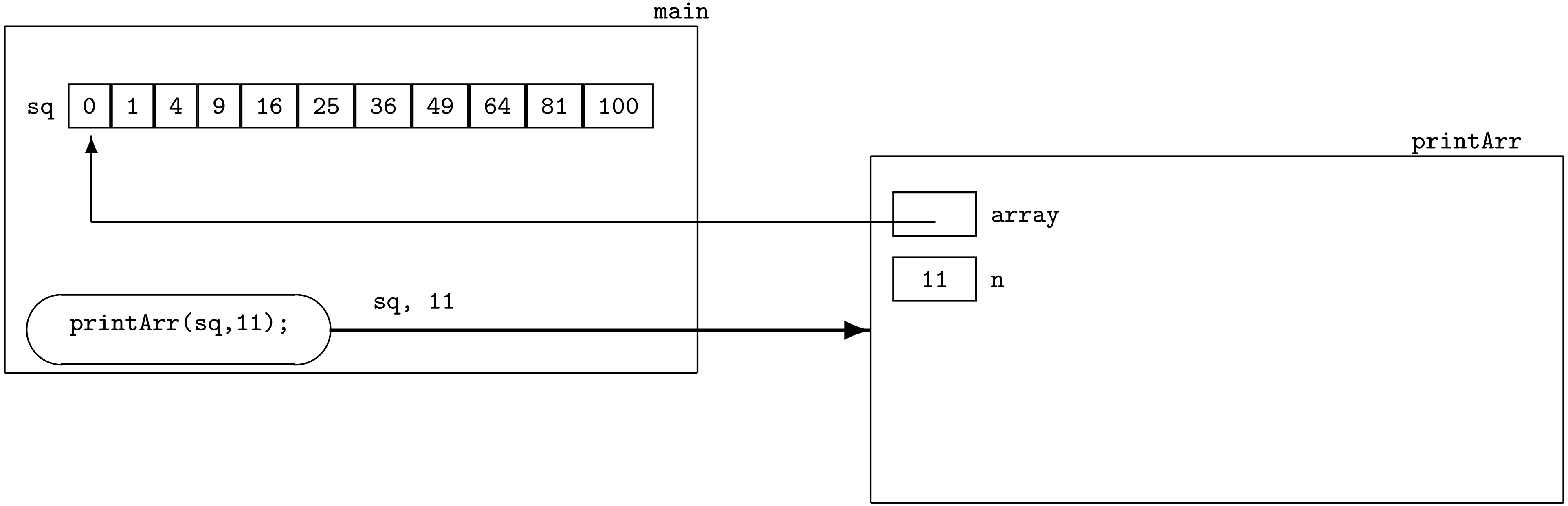
- the array (specifically, its location)
- the number of array elements to process

Array as Function Argument

- □ An array must be declared in the caller to be passed to the function via its output parameter, e.g.

  ```
  void printArr(int arr[], int n);
  ```

- □ The array is shared between the two functions
- □ No need to specify size for the array output parameter



Example: Cumulative Sum

- The cumulative sum of the sequence $\{a, b, c, \ldots\}$ is given by $\{a, a+b, a+b+c, \ldots\}$.

```c
#include <stdio.h>

#define SIZE 100

void readData(int arr[], int n);
void printArr(int arr[], int n);
void cumulSum(int arr[], int n);

int main(void) {
   int arr[SIZE], n;

   scanf("%d", &n);
   readData(arr, n);
   cumulSum(arr, n);
   printArr(arr, n);
   return 0;
}
```

```c
void readData(int arr[], int n) {
   int i;

   for (i = 0; i < n; i++) {
      scanf("%d", &(arr[i]));
   }
   return;
}

void printArr(int arr[], int n) {
   int i;

   for (i = 0; i < n; i++) {
      printf("%d ", arr[i]);
   }
   printf("\n");
   return;
}
```

Example: Cumulative Sum

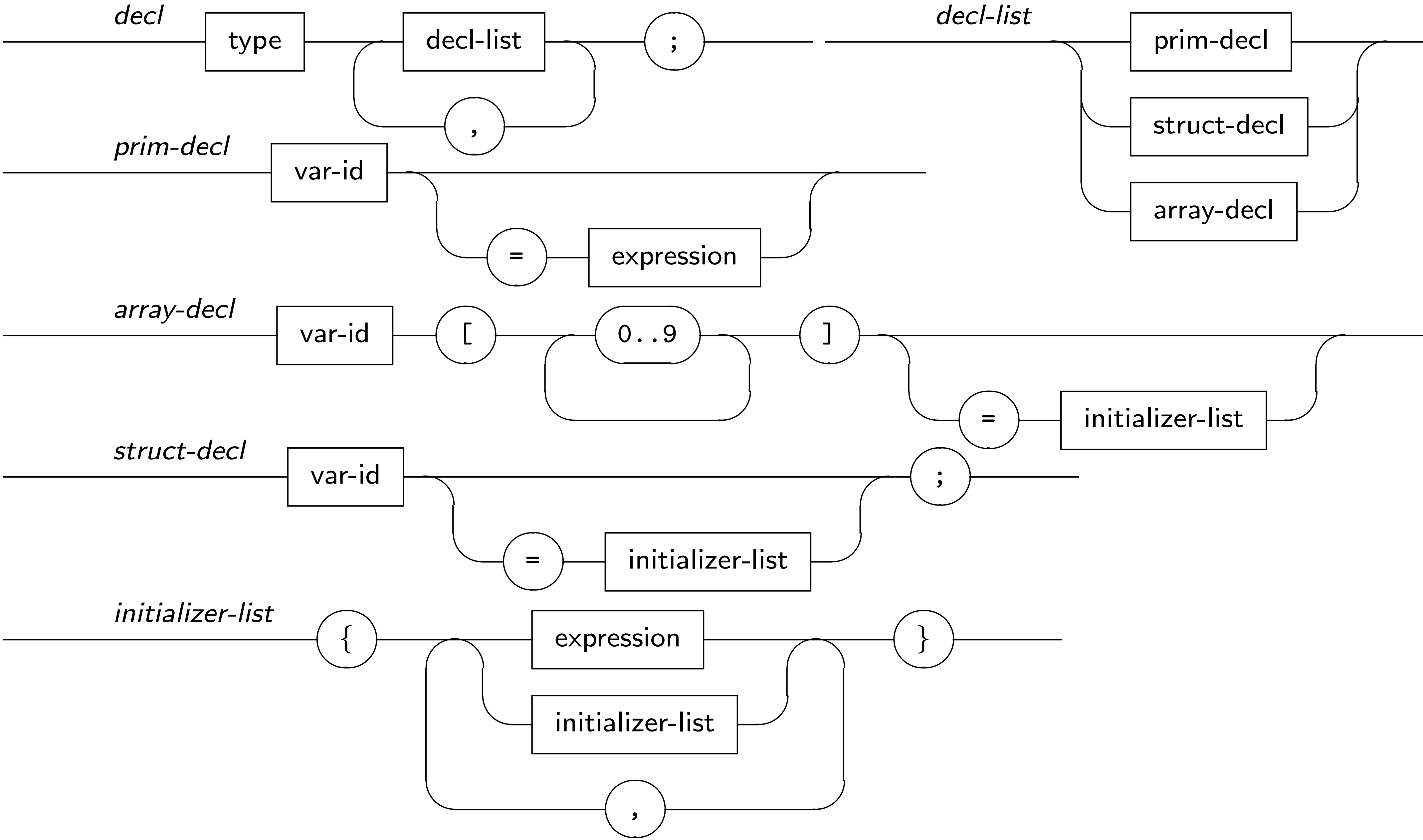
```c
void cumulSum(int arr[], int n) {
   int i;

   for (i = 1; i < n; i++) {
      arr[i] = arr[i] + arr[i - 1];
   }
   return;
}
```

- Note that `i` loops from 1 to `n - 1`
- Another way to express the loop

```c
for (i = 0; i < n - 1; i++) {
   arr[i + 1] = arr[i + 1] + arr[i];
}
```

- Ensure that array access stays within bounds

Declaration Revisited



decl

type

decl-list

,

;

decl-list

prim-decl

struct-decl

array-decl

prim-decl

var-id

=

expression

array-decl

var-id

[

0..9

]

=

initializer-list

struct-decl

var-id

=

initializer-list

;

initializer-list

{

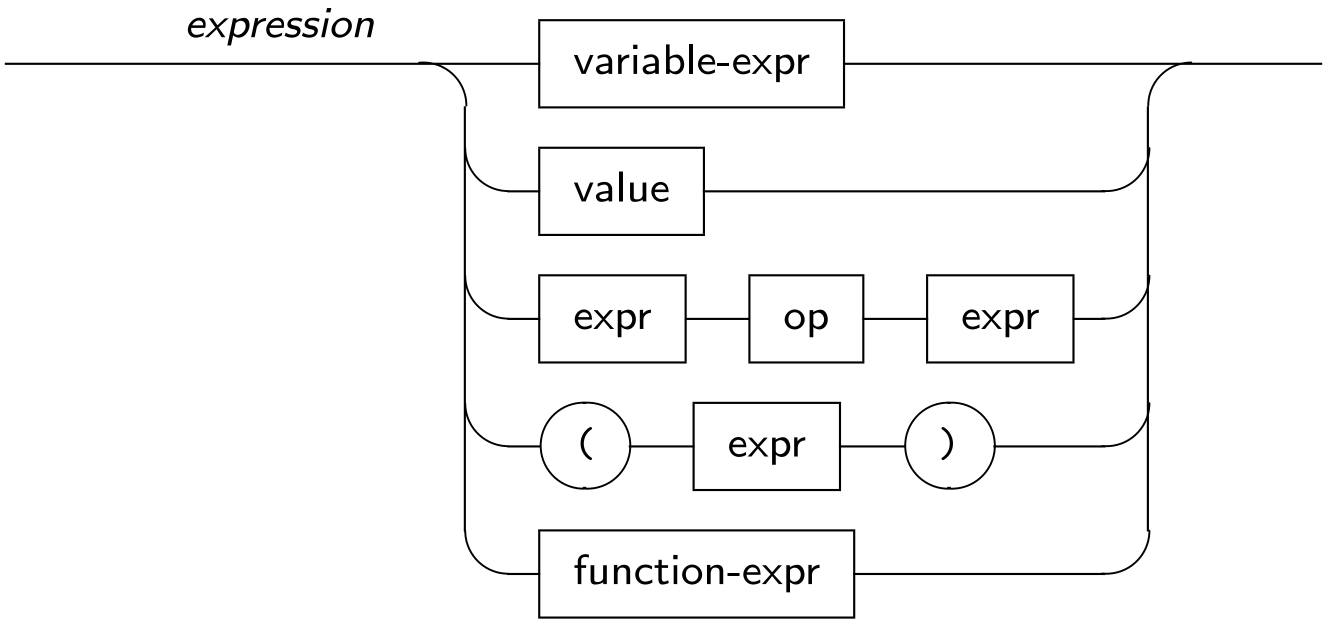
expression

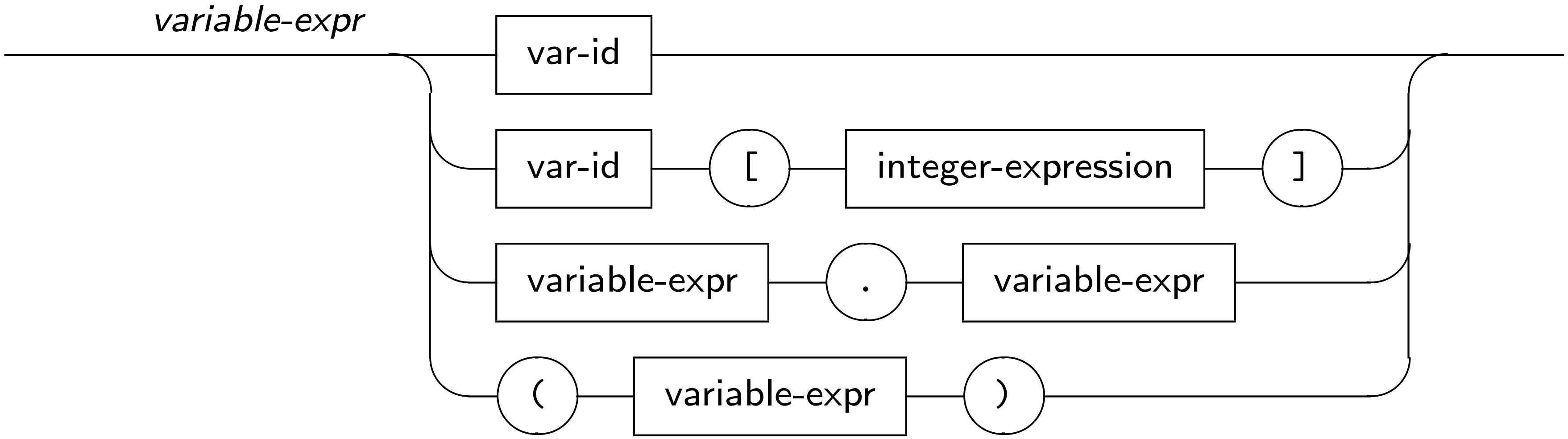
initializer-list

,

}

Expression Revisited





Arrays with Structures

- □ Array of structures

```
Point points[3] = { {0,0}, {1,1}, {2,2} };
```

- □ Structure with array as member

```
typedef struct {
   int numDim;
   double pt[MAX_DIM];
} MultiPoint;

Multipoint point4D = { 4, {1.9, 2.8, 3.7, 4.6} };
```

- □ Structure with structure as member

```
typedef struct {
   Point bottomleft, topright;
} Rectangle;

Rectangle rect = { {1.0, 2.0}, {3.0},{4.0} };
```

- □ Array of arrays (multi-dimensional arrays)

Lecture Summary

□ Structures

- Understand that structures behave similarly to primitives
- Use of structure member operator `.` to access individual members
- Use of structures for pass-by-value and return value in functions

□ Arrays (one dimensional)

- Appreciate that an array must be declared/initialized with a pre-determined size
- Use of subscripting/indexing to assess individual elements
- Use of loops (typically `for` loop) to access array elements
- Able to pass arrays via function output parameters