**Lecture 2:**

- Overview of MyPL Continued

**Homework:**

- HW-0 out (setup)

## 4. **Variable declarations**: must have types, can have initializers

```
int x1;                           // declares x1, initialized to null
int x2 = 5;                       // declares x2, initialized to 5
int x3 = 5 * 3 + 2;               // declares x3, initialized to 17
double y1;                        // declares y1, initialized to null
double y2 = 3.14159;              // declares y2, initialized to value
bool flag = true;                 // declares flag, initialized
int x4 = x1 + x2;                 // runtime error because of null + 5
string s1;                        // declares s1, initialized to null
string s2 = "Hello";              // declares s2, initialized to value
string s3 = null;                 // declares s3, initialized to null
int x5 = null;                    // declares x5, initialized to null
```

## 5. **Variable assignments**: require defined vars and matching types

```
    x = 10;
    z = false;
    u = 3.1 * 4.2 + v;
    w = null;              // works for any type
```

Notes on `null`:

- for initialization, assignment, and comparison (`!=`, `==`)
- other uses result in a runtime error (e.g., `5 + null` or `x < null`)

## 6. **Relational comparators and Boolean operations**

- normal comparators: >, <, <=, >=, !=, == ... for most primitive types
- normal Boolean operators: `and`, `or`, `not`
- expressions can be parenthesized, e.g., `(x <= y) or ((x == 0) and (y > 1))`

7. **For loops**: three parts – variable definition, condition, assignment

```
int x = 0

for (int i = 1; i <= 5; i = i + 1) {
  x = x + i;
}
int y = x;                    // y is 15, i undefined

for (int j = 5; j >= 1; j = j - 1) {
  y = y - j;
}
int z = y;                    // z is 0, j undefined
```

8. **While loops**: normal version of a while loop

```
while (i > 0 and i < 10) {
  ...
}
```

9. **If statements**: normal version of if-elseif-else statements

```
if (x == 1 or x == 2) {
   ...
}
elseif (y > 20 and y <= 30) {
   ...
}
elseif (y > 30) {
  ...
}
else {
   ...
}
```

## 10. **Arithmetic operations**: used only over int and double values

- can be parenthesized and chained together (e.g., `x + (y * z)`)
- no type cohersion (only add ints, only add doubles, etc., no mixing)

```
x + y     // int/double addition, string concatenation
x - y     // int/double subtraction
x * y     // int/double multiplication
x / y     // int/double division
```

## 11. **Functions**: return types, param types, recursion allowed

```
int f(int x, int y) {
  int z = x + y;
  if (x < y) {
    z = 0 - z;
  }
  return z;
}

// print is a special one-argument "built-in" function
void g(int x) {
  print(x);
  print("\n");
}

// nth fibonacci number
int fib(int n) {
  if (n < 0) {
    return null;
  }
  if (n == 0 or n == 1) {
    return n;
  }
  return fib(n-1) + fib(n-2);
}

// examples of function calls
int x = f(1, 2);   // call f
g(x);              // call g
int y = fib(10);   // call fib
```

12. **Structs (aka record) types**: collection of variables forming objects

- dynamically allocated (on the heap)
- on allocation, each variable (field) given initial value ("constructor")

```
struct Node {
  int val;
  Node next;
}

Node n1 = new Node(10, null);        // create object on the heap
Node n2 = new Node(20, null);        // create object on the heap
n1.next = n2;                        // link n2 to n1
n2.next = new Node(30, null);        // create Node linked to n2
n2.next.next = new Node(40, null);   // create another Node
bool eq1 = n1 == n2;                 // false, different references
Node n5 = n2.next.next.next;         // n5 is null
bool eq2 = n5 == null;               // true

int v = n5.val                       // runtime error
```

13. **Arrays**: fixed size, indexed collection of values of the same type

- dynamically allocated (on the heap)
- the expression "**new** $t$[$n$]" creates an array with $n$ null values

```
array int xs;                              // xs is null, type is int array
array int ys = new int[10];                // ys is 10 element int array
ys[2] = 42;                                // assign third element
int y = ys[2];                             // get third element

array Node ys = new Node[y]                // array of linked-list nodes

// function that returns an array
array int init(int length, int value) {
  array int tmp = new array[length]
  for (int i = 0; i < length; i = i + 1) {
    tmp[i] = value;
  }
  return tmp;
}

// function that takes an array
int get(array int xs, int i, int or_value) {
  if (i >= 0 and i < length(xs)) {
    return xs[i];
  }
  else {
    return or_value;
  }
}
```

*Note on deallocation:* no garbage collection or delete (could be a final project)

## 14. Automatic type conversion not supported

- for example, the following result in **type errors**:

```
int x = "4";         // type error (expected int, found string)
bool y = 1;          // type error (expected bool, found int)
double z = 3;        // type error (expected double, found int)
```

## 15. Built-in functions for type conversion

```
int x1 = dtoi(3.14);          // double to int
int x2 = stoi("4");           // string to int

double x3 = itod(4);          // int to double
double x4 = stod("3.14");     // string to double

string x5 = itos(4);          // int to string
string x6 = dtos(3.14);       // double to string
```

- `stoi()`, `stod()` can result in runtime errors

## 16. Additional built-in functions

```
print("Hello World\n");        // print any base type value
string y1 = input();           // read string from standard in
int y2 = length("foo");        // string or array length
string y4 = get(0, "foo");     // get i-th string char
```

## 17. MyPL Programs

- only consider single-file programs (in a text file, but we use `.mypl` extension)
- a MyPL file consists of struct and function definitions
- normal scoping rules (more later)
- must have a **void main()** function (called when program is run)

*Example 1: Simple Struct*

```
struct Car {
  string make;
  string model;
  int year;
}

void print_car(Car c) {
  print(c.make + " " + c.model + " " + itos(c.year) + "\n");
}

void main() {
  Car c1 = new Car("Toyota", "Corolla", 2023);
  Car c2 = new Car("Honda", "Civic", 2022);
  print_car(c1);
  print_car(c2);
}
```

*Example 2: Simple Linked List*

```
struct Node {
  int val;
  Node next;
}

void main() {
  // creates a linked list: 10, 20, 30, 40, 50
  Node head = null;
  int len = 5;
  for (int i = 0; i <= (len - 1); i = i + 1) {
    head = new Node((len - i) * 10, head);
  }
  // print the list
  Node curr = head;
  int i = 0;
  while (i < len) {
    print(curr.val);
    if (i < (len - 1)) {
      print(", ");
    }
    curr = curr.next;
    i = i + 1;
  }
  print("\n");
}
```

*Exasmple 3: Catalan Numbers*

```
int fac(int n) {
  if (n <= 0) {
    return 1;
  }
  return r = n * fac(n - 1);
}

int catalan_number(int n) {
  if (n < 0) {
    // only defined for n >= 0
    return 0;
  }
  return fac(2 * n) / (fac(n + 1) * fac(n));
}

// prints: 1, 1, 2, 5, 14, 42, 132, ...
void main() {
  print("Enter the number of catalan numbers to print: ")
  int m = stoi(input());
  for (int n = 0; n < m; n = n + 1) {
    int c = catalan_number(n);
    print("Catalan number " + itos(n) + " = " + c + "\n");
  }
}
```

## ***Summary – Things to Know***

1. How to write a MyPL function that computes the sum of a given array of integer values.

2. How to write a MyPL program that repeatedly prompts a user for a number until they enter "-1", and then prints out the sum of the numbers they entered (excluding -1).

3. How to write a MyPL program that implements a recursive function. A good one to try is mergesort over integer arrays.

4. How to implement a "stack" (e.g., with functions to push, pop, peek) in MyPL using a linked list data structure.