Pointer and Reference Types

- Pointer type values consists of memory addresses

- Pointers have been designed for the following uses:

  - Allocate memory for variable and return a pointer to that memory.
  - Dereference a pointer to obtain the value of variable it points to.

  - Deallocate the memory of variable pointed to by a pointer.
  - Pointer arithmetic.

- Pointers improve writability
Doing dynamic data structures (linked lists, trees) in a language with no pointers (FORTRAN 77) must be emulated with arrays, which is very cumbersome.

Pointers in C++

The variable that stores the reference to another variable is what we call a *pointer*.

- & is the reference operator and can be read as "address of"
- * is the dereference operator and can be read as "value pointed by"

```
#include <iostream>
using namespace std;

int main () {
  int firstvalue = 5, secondvalue = 15;
  int * p1, * p2;
  p1 = &firstvalue;  // p1 = address of firstvalue
  p2 = &secondvalue; // p2 = address of secondvalue
  *p1 = 10;          // value pointed by p1 = 10
  *p2 = *p1;         // value pointed by p2 = value pointed by p1
  p1 = p2;      // p1 = p2 (value of pointer is copied)
  *p1 = 20;          // value pointed by p1 = 20
  cout << "firstvalue is " << firstvalue << endl;
  cout << "secondvalue is " << secondvalue << endl;
  return 0;
}
```

```
firstvalue ?
secondvalue?
```

Pointers and arrays

- The identifier of an array is equivalent to the address of its first element, as a pointer is equivalent to the address of the first element that it points to.

```
int numbers [20];
int * p;
```

The following assignment operation would be valid:

```
p = numbers;
```

Unlike p, which is an ordinary pointer, numbers is an array, and an array can be considered a *constant pointer*. Therefore, the following allocation would not be valid:

```
numbers = p;
```

Exercise:

```
// more pointers
#include <iostream>
using namespace std;

int main ()
{
  int numbers[5];
  int * p;
  p = numbers;  *p = 10;
  p++;  *p = 20;
  p = &numbers[2];  *p = 30;
  p = numbers + 3;  *p = 40;
  p = numbers;  *(p+4) = 50;
  for (int n=0; n<5; n++)
    cout << numbers[n] << ", ";
  return 0;
}
```

Pointer arithmetics

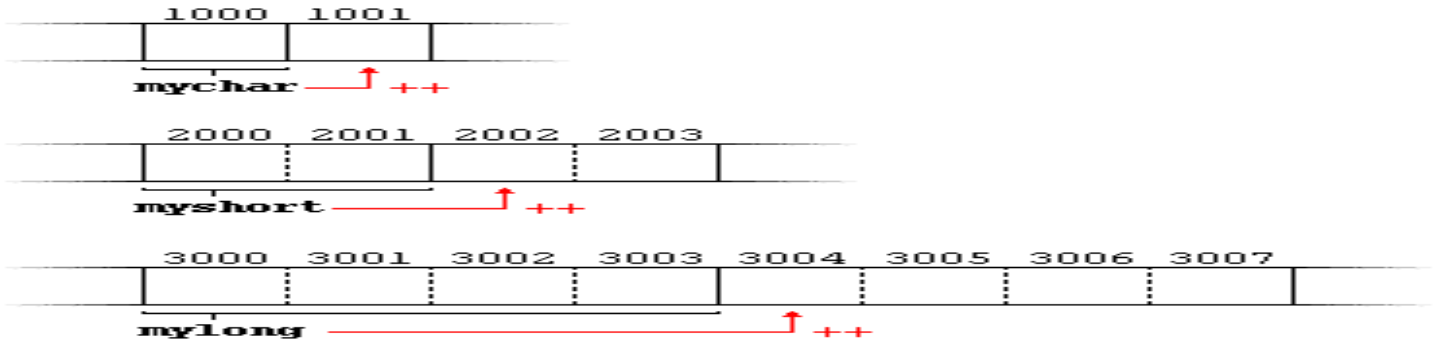
When we saw the different fundamental data types, we saw that some occupy more or less space than others in the memory.

char takes 1 byte, short takes 2 bytes and long takes 4.

```
char *mychar;
short *myshort;
long *mylong;
```

and that we know that they point to memory locations 1000, 2000 and 3000 respectively.

```
mychar++;
myshort++;
mylong++;
```



The following expression may lead to confusion:

```
*p++
```

Because ++ has greater precedence than *, this expression is equivalent to *(p++).

Notice the difference with:

(*p)++

If we write:

```
*p++ = *q++;
```

Because ++ has a higher precedence than *, both p and q are increased, but because both increase operators (++) are used as postfix and not prefix, the value assigned to *p is *q before both p and q are increased. And then both are increased.

It would be roughly equivalent to:

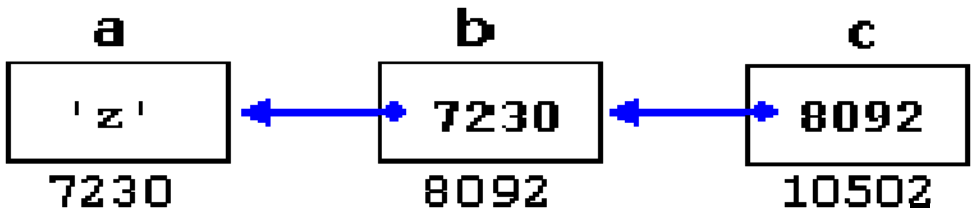
```
*p = *q;
++p;
++q;
```

Pointers to pointers

C++ allows the use of pointers that point to pointers

```
Char a;
char * b;
char ** c;
a = 'z';
b = &a;
c = &b;
```

Supposing the randomly chosen memory locations for each variable of 7230, 8092 and 10502,



- c has type char** and a value of 8092
- *c has type char* and a value of 7230
- **c has type char and a value of 'z'

void pointers

- A special type of pointer. In C++, void pointers are pointers that point to a value that has no type.

- This allows void pointers to point to any data type, from an integer value or a float to a string of characters.

- Limitation: the data pointed by them cannot be directly dereferenced, and for that reason we will always have to change the type of the void pointer to some other pointer type that points to a concrete data type before dereferencing it.

  This is done by performing type-castings.

One uses of void pointer may be to pass generic parameters to a function:

```
// increaser
#include <iostream>
using namespace std;

void increase (void* data, int size)
{
  switch (size)
  {
    case sizeof(char) : (*((char*)data))++; break;
    case sizeof(int) : (*((int*)data))++; break;
  }
}

int main ()
{
  char a = 'x';
  int b = 1602;
  increase (&a,sizeof(a));
  increase (&b,sizeof(b));
  cout << a << ", " << b << endl;
  return 0;}
```

```
y, 1603
```

Null pointer

- It is not pointing to any valid reference or memory address.

```
int * p;
```

Difference between void pointer and null pointer

- A null pointer is a value that any pointer may take to represent that it is pointing to "nowhere",
- A void pointer is a special type of pointer that can point to somewhere without a specific type.

Pointers to functions

Function Pointers are pointers that point to the address of a function.

In order to declare a pointer to a function we have to declare it like the prototype of the function except that the name of the function is enclosed between parentheses () and an asterisk (*) is inserted before the name:

```
// pointer to functions
#include <iostream>
using namespace std;

int addition (int a, int b)
{ return (a+b); }

int subtraction (int a, int b)
{ return (a-b); }

int operation (int x, int y, int (*functocall)(int,int))
{
  int g;
  g = (*functocall)(x,y);
  return (g);}
```

```
int main ()
{
  int m,n;

  m = operation (7, 5, addition);
  n = operation (20, m, subtraction);
  cout <<n;
  return 0;
}
```

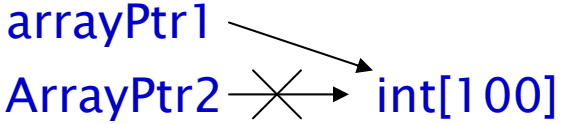
Problems with Pointers

- PL/I was the first programming language to provide a pointer type.
- Pointers come with several problems (PL/I, C, C++, etc.)

(1) Dangling pointers (dangerous)

–A pointer points to a heap-dynamic variable that has been de-allocated

In C++.




```
int * arrayPtr1;
int * arrayPtr2 = int[100];
arrayPtr1 = arrayPtr2;
delete [] arrayPtr2;
// Now arrayPtr1 is dangling.
```

(2) Lost heap-dynamic variable

- An allocated heap-dynamic variable that is no longer accessible to the user program (often called *garbage*)

```
int * Ptr1=  10;
int * Ptr2 = 20;
Ptr1 = Ptr2;
```

(3) pointers can be misused is to access outside the data structure they point to.

```
#include <stdio.h>
#include <stdlib.h>

int main(void){
   int y = 5;                          /* create a variable  */
   int *p1 = &y;                          /* initialize pointer to y */
   printf("Value of  *p1: %d\n", *p1);        /* value of y */
   p1 = p1 + y;                          /* allowed pointer arithmetic */
   printf("Value of  *p1: %d\n", *p1);        /* p1 no longer points to y */
   return 0;}
```

Evaluation of Pointers

- Dangling pointers and Lost heap-dynamic variable are problems
- Pointers or references are necessary for dynamic data structures

Reference Types of C++

- A C++ reference type variable is a constant, it must be initialized with the address of some variable in its definition, and after initialization a reference type variable can never be set to reference any other variable.

```
int num1 = 10;
int num2 = 20;

int &RefOne = num1;        // valid
int &RefOne = num2;        // error, two definitions of RefOne
int &RefTwo = num2;        // valid
```

- In C++, the real usefulness of references is when they are used to pass values into functions.

C++ references differ from pointers in several essential ways:

- Neither arithmetic, casts, nor any other operation can be performed on references.
- Once a reference is created, it cannot be later made to reference another object. This is often done with pointers.
- References cannot be *null*, whereas pointers can; every reference refers to some object, although it may or may not be valid.

```
#include <iostream>
using namespace std;

int main(){
   int val = 1;
   int &rVal = val;
   cout << "val is " << val << endl;
   cout << "rVal is " << rVal << endl;
   cout << "Setting val to 2" << endl;
   val = 2;
   cout << "val is " << val << endl;
   cout << "rVal is " << rVal << endl;
   return 0;
}
```

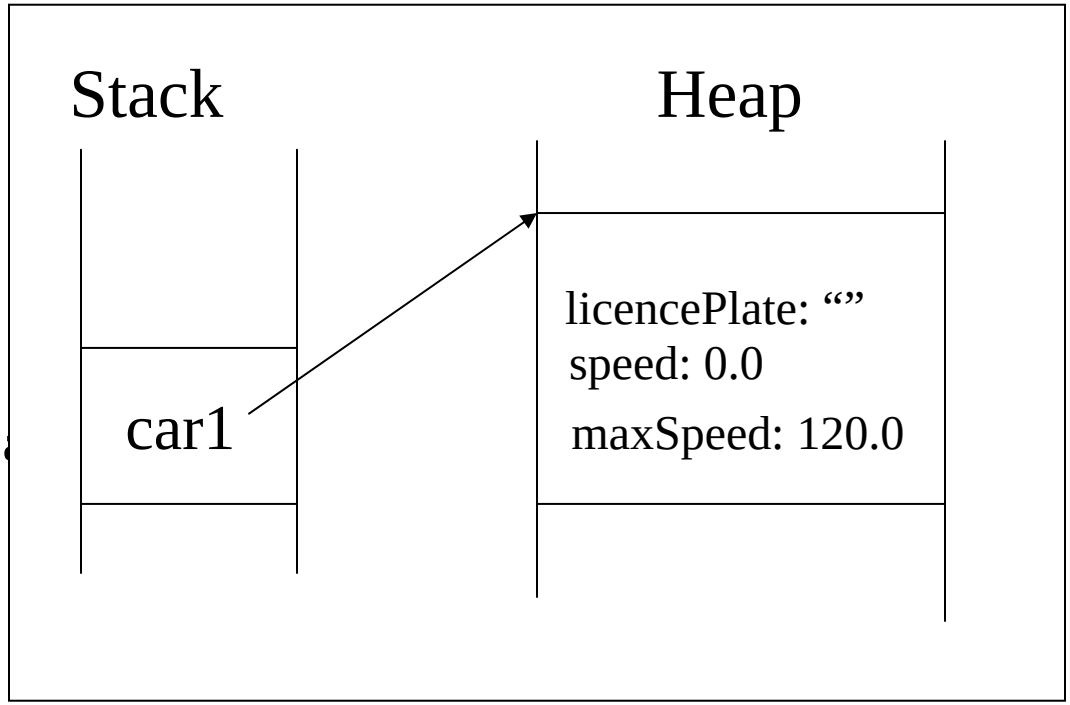
Reference Types of Java

- Increases safety over C++. The fundamental difference between C++ pointer and Java reference is that C++ pointers refer to memory addresses, whereas Java reference to objects. This immediately prevents arithmetic on references from being sensible.
- Unlike reference in C++, Java does not disallow assignment, Java reference variables can be assigned to refer to different objects.
- Java objects are implicitly deallocated, there cannot be a dangling reference.

Reference in Java

Car car1 = new Car ();

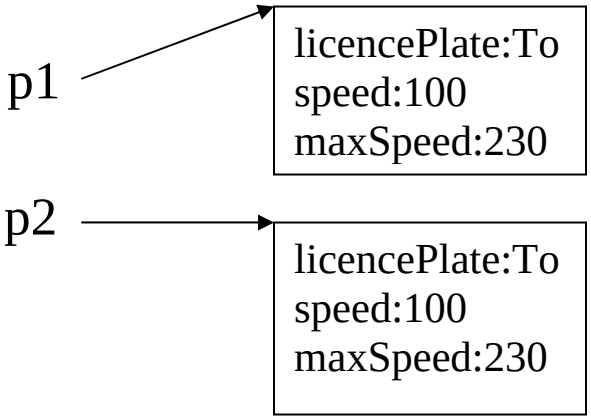
```
class Car {
  String licensePlate = “”;
  double speed       = 0.0;
  double maxSpeed    = 120.0;
}

class Cartest {
    public static void main(String[] a
          ……
   Car car1 = new Car();
          .......
}
```



Garbage Collection

• When an object no longer has any valid references to it, it can no longer be accessed by the program.

```
class Car {
  String licensePlate = "To";
  double speed = 100;
  double maxSpeed = 230;
  double GetSpeed () {
        return speed;
  }

  void SetSpeed () {
        speed = 150;
  }
}
```



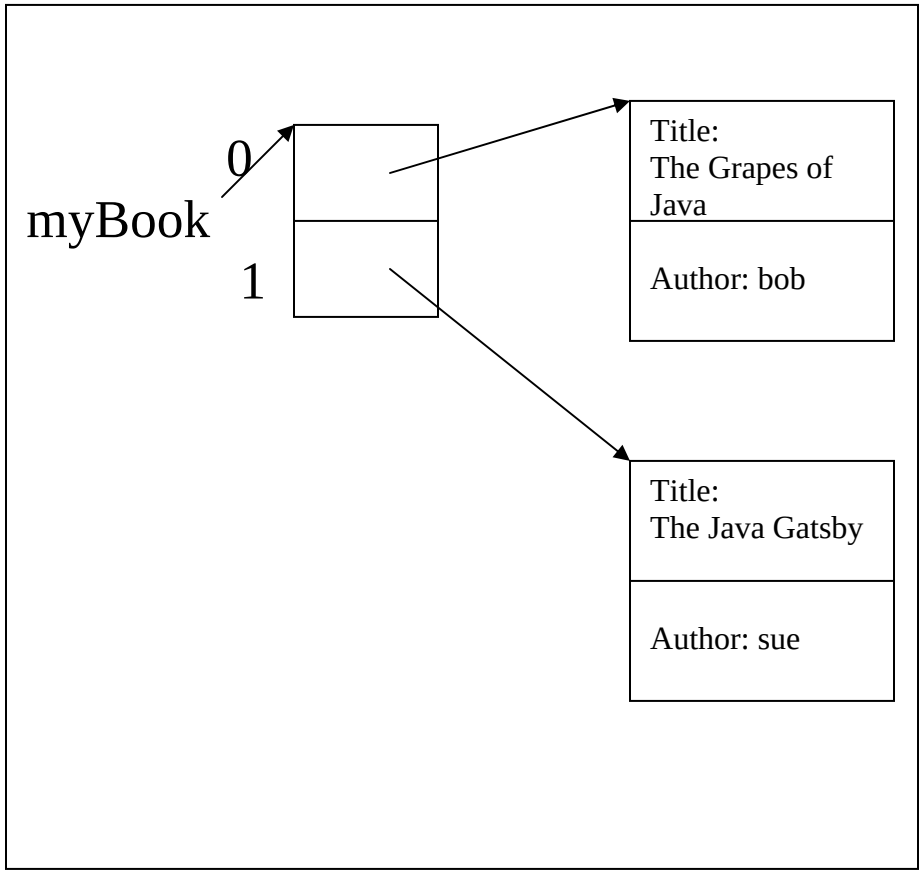
What will be the output when the following code is executed:

```
Car p1 = new Car ();
Car p2 = new Car ();
p1.SetSpeed();
p2=p1;
System.out.println (p2.GetSpeed());
```

Which object is garbage after the execution of the program?

Arrays: That type can be primitive types or objects

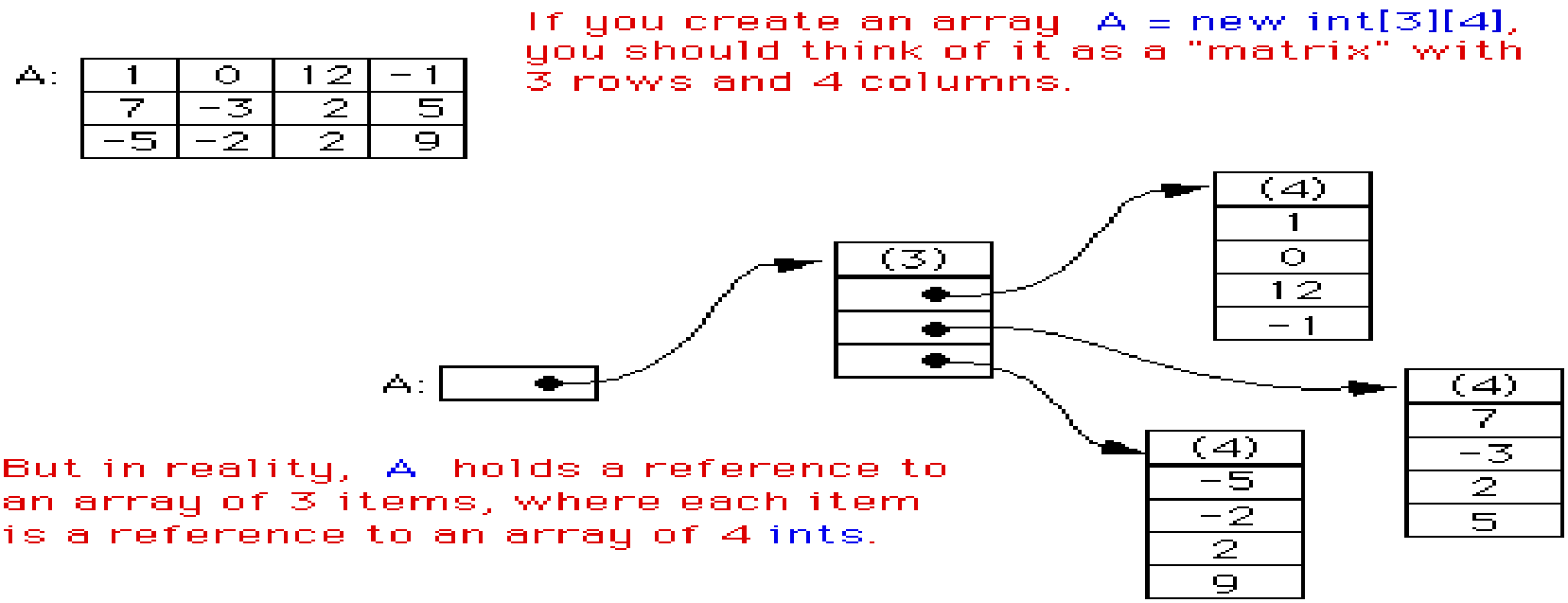
```
Class Books {
      String title;
      String author;
}

class BooksTestDrive {
    public static void main (String [] args) {
        Books [] myBooks = new Books[2];
        int x=0;

       myBooks[0].title = “The Grapes of Java ”;
       myBooks[0].author = “bob”;
       myBooks[1].title = “The Java Gatsby ”;
       myBooks[1].author = “sue”;

     While (x<2) {
              System.out.print (myBook[x].title);
              System.out.print (“ by ”);
              System.out.println (myBooks [x].author);
              x = x+1;
              }}}
```



Two-Dimensional Arrays

In Java a two-dimensional array is an array of arrays

```
int[][] A = new int[3][4];
```



- **A array element is referenced using two index values:**

```
int value = A[1][1]      ?
```

Initialize multi-dimensional array with specified items at the time it is declared.

```
int[][]  A  =  {  {  1,  0, 12, -1 },
                  {  7, -3,  2,  5 },
                  { -5, -2,  2,  9 }
               };
```

If no initializer is provided for an array,

int[][] A = new int[3][4];

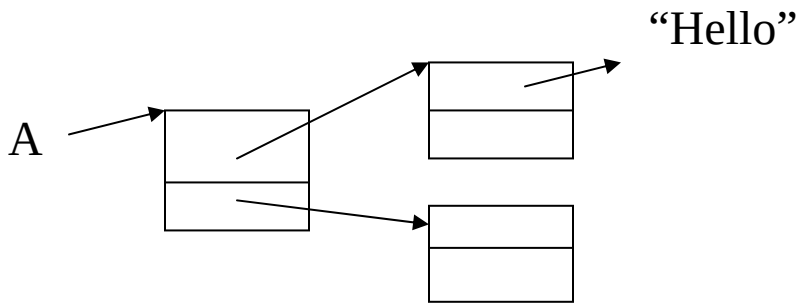
then zero for numbers, false for boolean, and null for objects.

int[][] A = new int[3][4];

A.length: the number of rows of A.

A[0].length: the number of columns in A
A[1].length
A[2].length

```
public class 2DArray
{
  public static void main (String[] args)
 {
  String [][] A = {
                    {"Hello", "World" },
                    {"Guten Tag", "Welt"}
                  };

  for (int row=0; row < A.length; row++)
    {
      for (int col=0; col < A[row].length; col++)
        System.out.println (A[row][col]);
    }

    }
  }
```



What would the following program display to standard output?

```
#include <stdio.h>
int do_stuff(int, int *, int *, int);
int a = 5;
int main() {
        int b = 10, c = 15, d = 20, e = 25, f;
        f = do_stuff(b, &c, &d, a);
        printf("%d, %d, %d, %d, %d, %d\n", a, b, c, d, e, f);
        return 0;
}

int do_stuff(int b, int *p1, int *p2, int x) {
        int e;
        e = 30;
        a = 35;
        x = 40;
        *p1 = 45;
        b = 50;
        p2 = &b;
        *p2 = 55;
        return b;
}
```

Answer:
35, 10, 45, 20, 25, 55

Before call do_stuff()

a = 5
b=10
c = 15
d = 20
e = 25
f = ?

After call do_stuff()

a = 35
b=10
c = 45
d = 20
e = 25
f = 55