Pointers and its concepts

From variables to their addresses

Basics of Pointers

Introduction

A pointer is a variable that represents the location (rather than the value) of a data item.

They have a number of useful applications.

- Enables us to access a variable that is defined outside the function.
- Can be used to pass information back and forth between a function and its reference point.

Basic Concept

In memory, every stored data item occupies one or more contiguous memory cells.

- The number of memory cells required to store a data item depends on its type (char, int, double, etc.).

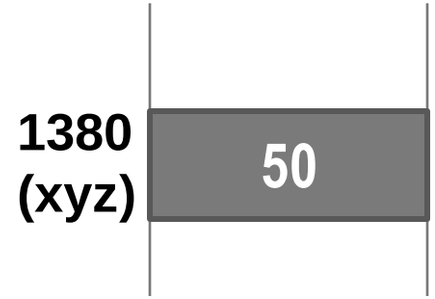
Whenever we declare a variable, the system allocates memory location(s) to hold the value of the variable.

- Since every byte in memory has a unique address, this location will also have its own (unique) address.

Example

Consider the statement

```
int  xyz = 50;
```



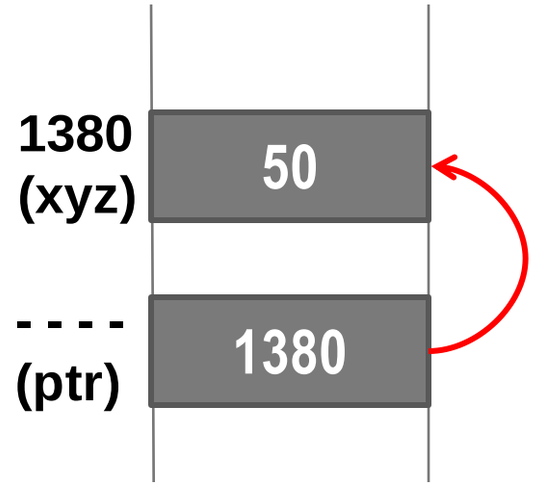
- This statement instructs the compiler to allocate a location for the integer variable xyz, and put the value 50 in that location.

- Suppose that the address location chosen is 1380.
- During execution of the program, the system always associates the name xyz with the address 1380.

- The value 50 can be accessed by using either the name xyz or the address 1380.

Example (Contd.)

```
int xyz = 50;
int *ptr;  // Here ptr is a pointer to an integer
ptr = &xyz;
```



Since memory addresses are simply numbers, they can be assigned to some variables which can be stored in memory.

- Such variables that hold memory addresses are called *pointers*.
- Since a pointer is a variable, its value is also stored in some memory location.

Pointer Declaration

A pointer is just a C variable whose value is the address of another variable!

After declaring a pointer:

```
int *ptr;
```

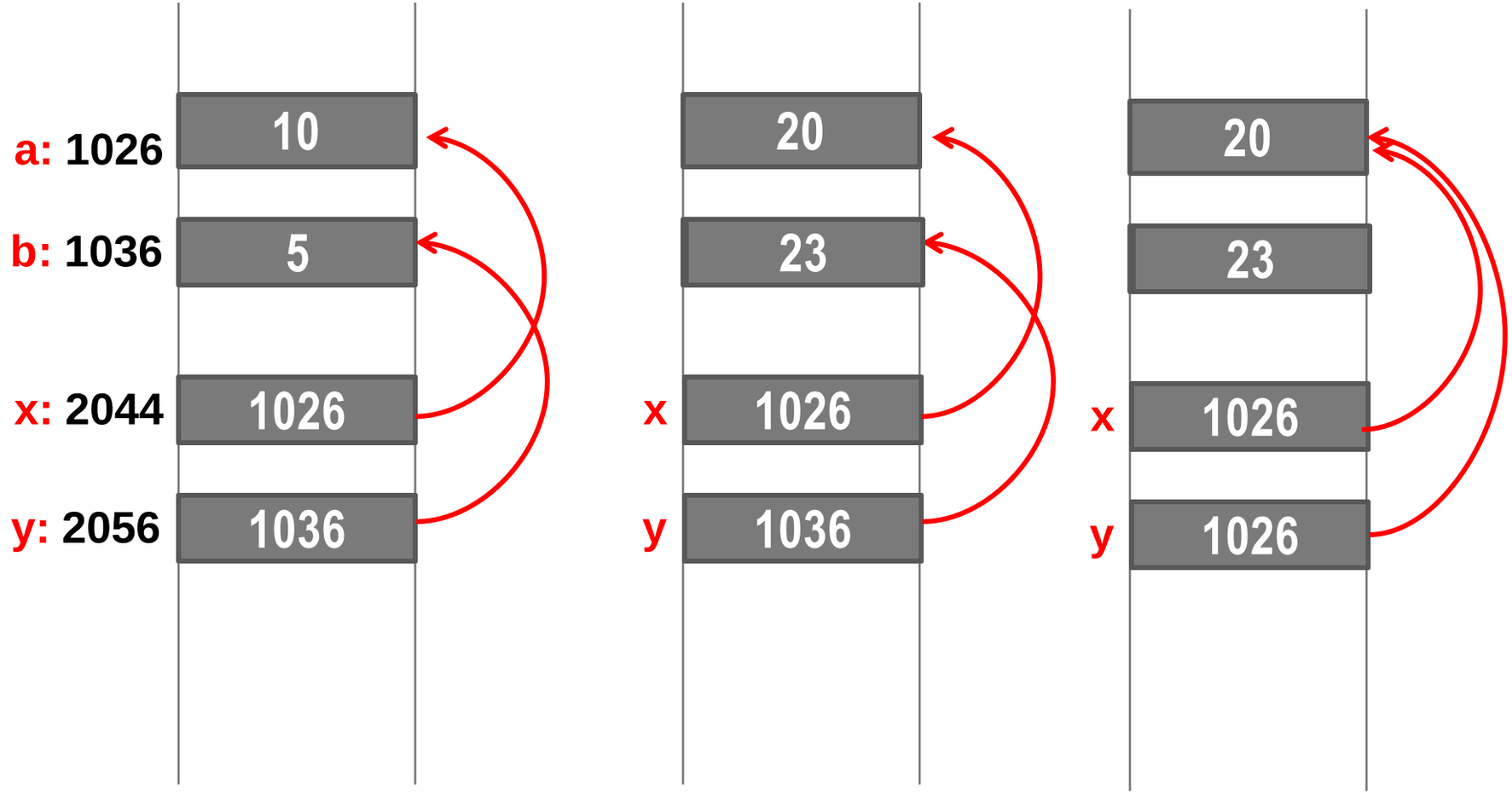
ptr doesn't actually point to anything yet.

We can either:

- make it point to some existing variable (which is in the stack), or
- dynamically allocate memory (in the heap) and make it point to it

Making it point

```
int a=10, b=5;
int *x, *y;
x= &a; y=&b;
*x= 20;
*y= *x + 3;
y= x;
```



Accessing the Address of a Variable

The address of a variable can be determined using the '&' operator.

- The operator '&' immediately preceding a variable returns the *address* of the variable.

Example:

```
p = &xyz;
```

- The *address* of xyz (1380) is assigned to p.

The '&' operator can be used only with a *simple variable* or an *array element*.

```
&distance
&x[0]
&x[i-2]
```

Illegal usages

Following usages are illegal:

```
&235
```

- Pointing at constant.

```
int   arr[20];
  :
&arr;
```

- Pointing at array name.

```
&(a+b)
```

- Pointing at expression.

Pointer Declarations and Types

Pointer variables must be declared before we use them.

General form:

```
data_type *pointer_name;
```

Three things are specified in the above declaration:

- **The asterisk (*) tells that the variable pointer_name is a pointer variable.**
- **pointer_name needs a memory location.**
- **pointer_name points to a variable of type data_type.**

Pointers have types

Example:

```
int   *count;
float *speed;
```

Once a pointer variable has been declared, it can be made to point to a variable using an assignment statement like:

```
int *p, xyz;
 :
 p = &xyz;
```

- **This is called *pointer initialization*.**

Things to remember

Pointer variables must always point to a data item of the *same type*.

```
float   x;
int    *p;
p = &x;          // This is an erroneous assignment
```

Assigning an absolute address to a pointer variable is prohibited.

```
int   *count;
count = 1268;
```

Pointer Expressions

Like other variables, pointer variables can be used in expressions.

If p1 and p2 are two pointers, the following statements are valid:

```
sum = (*p1) + (*p2);
prod = (*p1) * (*p2);
*p1 = *p1 + 2;
x = *p1 / *p2 + 5;
```

More on pointer expressions

What are allowed in C?

- Add an integer to a pointer.
- Subtract an integer from a pointer.
- Subtract one pointer from another
    - If p1 and p2 are both pointers to the same array, then p2–p1 gives the number of elements between p1 and p2.

More on pointer expressions

What are not allowed?

- **Add two pointers.**

```
p1 = p1 + p2;
```

- **Multiply / divide a pointer in an expression.**

```
p1 = p2 / 5;
p1 = p1 – p2 * 10;
```

Scale Factor

We have seen that an integer value can be added to or subtracted from a pointer variable.

```
int x[ 5 ] = { 10, 20, 30, 40, 50 };
int *p;

p = &x[1];
printf( "%d", *p);          // This will print 20

p++;                        // This increases p by the number of bytes for an integer
printf( "%d", *p);          // This will print 30

p = p + 2;                  // This increases p by twice the sizeof(int)
printf( "%d", *p);          // This will print 50
```

More on Scale Factor

```
struct  complex {
   float real;
   float imag;
};
struct complex x[10];

struct complex *p;

p = &x[0];                    // The pointer p now points to the first element of the array

p = p + 1;                    // Now p points to the second structure in the array
```

The increment of *p* is not by one byte, but by the size of the data type to which *p* points.

This is why we have many data types for pointers, not just a single “address” data type

Pointer types and scale factor

Data Type	Scale Factor
char	1
int	4
float	4
double	8

- If p1 is an integer pointer, then

  p1++

  will increment the value of p1 by 4.

Scale factor may be machine dependent

- The exact scale factor may vary from one machine to another.
- Can be found out using the `sizeof` function.

```
#include  <stdio.h>
main( )
{
  printf (“No. of bytes occupied by int is %d \n”,  sizeof(int));
  printf (“No. of bytes occupied by float is %d \n”,  sizeof(float));
  printf (“No. of bytes occupied by double is %d \n”, sizeof(double));
  printf (“No. of bytes occupied by char is %d \n”,   sizeof(char));
}
```

Output:

```
Number of bytes occupied by int is  4
Number of bytes occupied by float is  4
Number of bytes occupied by double is  8
Number of bytes occupied by char is  1
```

Passing Pointers to a Function

Pointers are often passed to a function as arguments.

- **Allows data items within the calling program to be accessed by the function, altered, and then returned to the calling program in altered form.**
- **Called *call-by-reference* (or by *address* or by *location*).**

Normally, arguments are passed to a function *by value*.

- **The data items are copied to the function.**
- **Changes are not reflected in the calling program.**

Passing arguments by value or reference

```
#include  <stdio.h>
main( )
{
    int  a, b;
    a = 5;  b = 20;
    swap (a, b);
    printf (“\n a=%d, b=%d”, a, b);
}

void  swap (int x, int y)
{
    int  t;
    t = x;  x = y;  y = t;
}
```

Output

a=5, b=20

```
#include  <stdio.h>
main( )
{
    int  a, b;
    a = 5;  b = 20;
    swap (&a, &b);
    printf (“\n a=%d, b=%d”, a, b);
}

void swap (int *x, int *y)
{
   int  t;
   t = *x;   *x = *y;   *y = t;
}
```

Output

a=20, b=5

Pointers and Arrays

When an array is declared:

- **The compiler allocates a *base address* and sufficient amount of storage to contain all the elements of the array in contiguous memory locations.**
- **The *base address* is the location of the first element (*index 0*) of the array.**
- **The compiler also defines the array name as a *constant pointer* to the first element.**

Example

Consider the declaration:

```
int x[5] = {1, 2, 3, 4, 5};
```

- Suppose that the base address of x is 2500, and each integer requires 4 bytes.

Element	Value	Address
x[0]	1	2500
x[1]	2	2504
x[2]	3	2508
x[3]	4	2512
x[4]	5	2516

Example (contd)

Both x and &x[0] have the value 2500.

p = x; and p = &x[0]; are equivalent

- We can access successive values of x by using p++ or p-- to move from one element to another.

Relationship between p and x:

p = &x[0] = 2500
p+1 = &x[1] = 2504
p+2 = &x[2] = 2508
p+3 = &x[3] = 2512
p+4 = &x[4] = 2516

*(p+i) gives the value of x[i]