CS1100 - Lecture 7

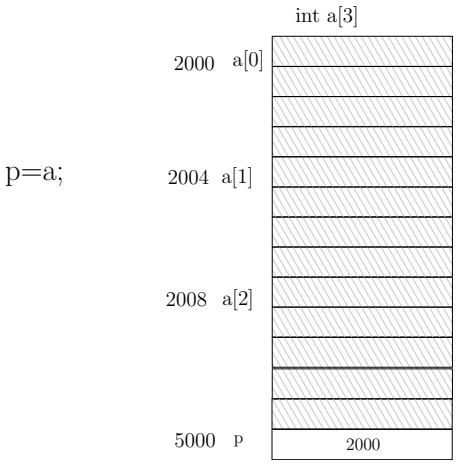
Instructor : Jasine Babu

Teaching Assistants : Nikhila K N, Veena Prabhakaran

Pointer arithmetic

Pointers support some arithmetic operations as well. If `p` is a pointer to an integer, the instruction `p=p+1` means, the current value of `p` (which is an address) is added with the number of locations needed to store an integer and the result is put as the new value of `p`. If `i` is an integer and `p` is a pointer to an integer, the expression `p+i` means the address obtained by adding `i` $\times$ `number of locations needed to store an integer` with current value of `p`. Similarly, `p-i` is also a valid expression. The instruction `p=p+i` means the value of `p` (which is an address) changes to `(current value of p) + (i` $\times$ `number of locations needed to store an integer)`.

In an array declaration like `int a[10]`, we already know the label `a` is not a variable, but it is equivalent to an address, which is the address of `a[0]`. Recall that, `a+i` is the address of `a[i]`. If we have an instruction `p=a` where, `p` is a pointer to an integer and `a` is an array of integers, then by the explanation given in previous paragraph, `a+i` and `p+i` are referring to the same address, which is, `&a[i]`.



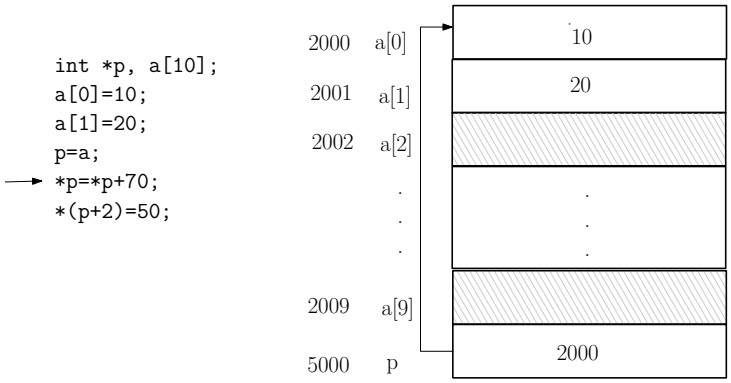
As per the figure given above, `p+2` is the address 2008, which is also `&a[2]` or `a+2`.

Now, let us consider an example program to understand the use of pointer arithmetic.

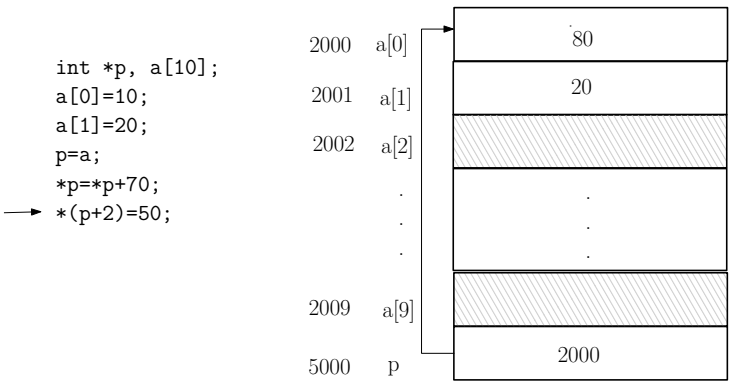
```
int *p, a[10];
a[0]=10;
a[1]=20;
p=a;
*p=*p+70;
*(p+2)=50;
```

Note that, the bracket on the left hand side of the last instruction is necessary because, unary operator `*` has higher precedence than `+,-,=,` etc. Without brackets `*p+2=50` is not a valid instruction. This is because,`*p+2` is only an integer value, not an address, and therefore it can not occur on the left hand side of an assignment instruction. Also note that, unary `*` and `&` are right associative.

After executing instructions upto `p=a` in the above program, the memory state diagram of the execution is shown below.

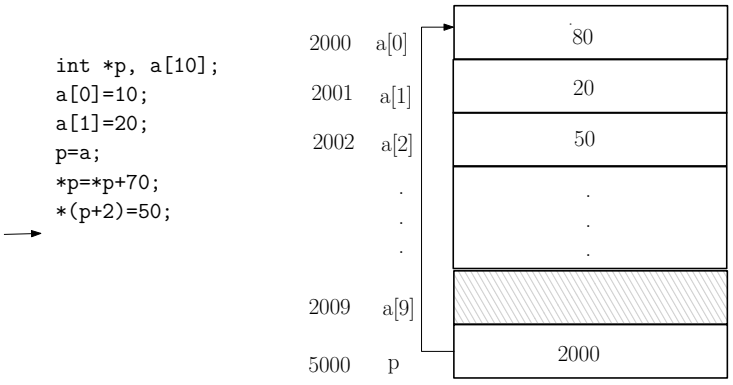


The next instruction to execute is `*p=*p+70`. Note that, the `*` operation has a higher precedence than the `+,=,-` operations. Now, since 2000 is the value of `p`, the expression `*p` refers to the location with address 2000 (i.e. the location of a[0]). Therefore, when the expression *p+10 is evaluated in CPU, the result is the sum of the current value stored in location 2000 and the value 10, which is equal to 10+70=80. After executing the instruction `*p=*p+70` the value stored in location 2000 gets updated to 80. As a result the value of `a[0]` changes to 80. The memory diagram after this stage is shown below.



Now, next instruction to execute is `*(p+2)=50`. Here, `p+2` is now same as the address of `a[2]` because, `p` and `a` denote the same value 2000. Therefore, `*(p+2)` refers to `a[2]`.

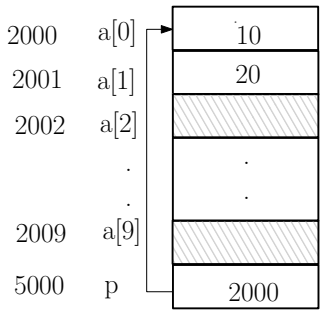
Hence, after executing `*(p+2)=50`, the value of `a[2]` changes to 50, as shown in the next figure.



The following example is to demonstrate the close connection between pointers and arrays in C.

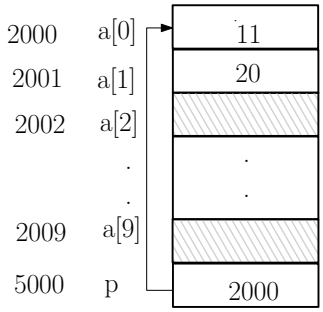
```
#include<stdio.h>
int main()
{
  int a[10], b, *p;
  a[0]=10;
  a[1]=20;
  p=a;
  *p = *p + 1;
  printf("a[0]=%d a[1]=%d *p=%d\n",a[0],a[1],*p);
  p=p+1;
  *p = *p + 1;
  printf("a[0]=%d a[1]=%d *p=%d\n",a[0],a[1],*p);
  *(p-1)=30;
  printf("a[0]=%d a[1]=%d *p=%d\n",a[0],a[1],*p);
  return(0);
}
```

After executing instructions upto `p=a;`, the memory state diagram is as shown below.




```
#include<stdio.h>
int main()
{
  int a[10], b, *p;
  a[0]=10;
  a[1]=20;
  p=a;
→ *p = *p + 1;
  printf("a[0]=%d a[1]=%d *p=%d\n",a[0],a[1],*p);
  p=p+1;
  *p = *p + 1;
  printf("a[0]=%d a[1]=%d *p=%d\n",a[0],a[1],*p);
  *(p-1)=30;
   printf("a[0]=%d a[1]=%d *p=%d\n",a[0],a[1],*p);
  return(0);
}
```

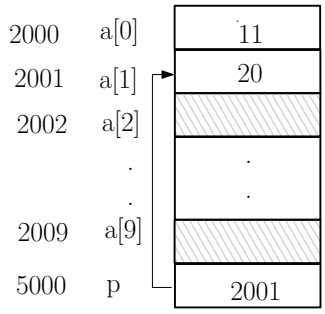
After executing instructions upto the first `*p=*p+1;`, the memory state diagram is as shown below.




```
#include<stdio.h>
int main()
{
  int a[10], b, *p;
  a[0]=10;
  a[1]=20;
  p=a;
  *p = *p + 1;
→ printf("a[0]=%d a[1]=%d *p=%d\n",a[0],a[1],*p);
  p=p+1;
  *p = *p + 1;
  printf("a[0]=%d a[1]=%d *p=%d\n",a[0],a[1],*p);
  *(p-1)=30;
   printf("a[0]=%d a[1]=%d *p=%d\n",a[0],a[1],*p);
  return(0);
}
```

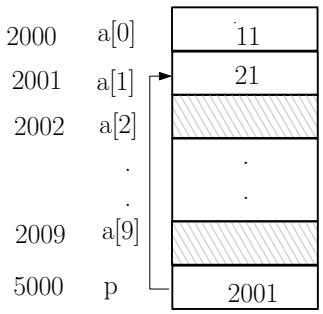
After execting first *printf* statement, output `a[0]=11 a[1]=20 *p=11` is displayed on screen.

After executing instructions upto `p=p+1;`, the memory state diagram is as shown below.




```
#include<stdio.h>
int main()
{
  int a[10], b, *p;
  a[0]=10;
  a[1]=20;
  p=a;
  *p = *p + 1;
  printf("a[0]=%d a[1]=%d *p=%d\n",a[0],a[1],*p);
  p=p+1;
→ *p = *p + 1;
  printf("a[0]=%d a[1]=%d *p=%d\n",a[0],a[1],*p);
  *(p-1)=30;
   printf("a[0]=%d a[1]=%d *p=%d\n",a[0],a[1],*p);
  return(0);
}
```

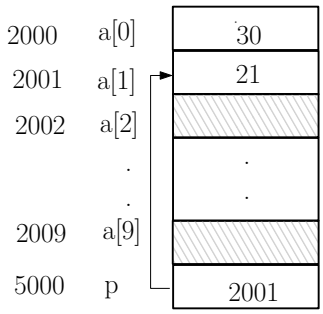
After executing instructions upto the second `*p=*p+1;`, the memory state diagram is as shown below.




```
#include<stdio.h>
int main()
{
  int a[10], b, *p;
  a[0]=10;
  a[1]=20;
  p=a;
  *p = *p + 1;
  printf("a[0]=%d a[1]=%d *p=%d\n",a[0],a[1],*p);
  p=p+1;
  *p = *p + 1;
→ printf("a[0]=%d a[1]=%d *p=%d\n",a[0],a[1],*p);
  *(p-1)=30;
   printf("a[0]=%d a[1]=%d *p=%d\n",a[0],a[1],*p);
  return(0);
}
```

After executing second *printf*, output `a[0]=11 a[1]=21 *p=21` is displayed on screen.

The memory state diagram after executing instructions upto `*(p-1)=30;` is shown below.



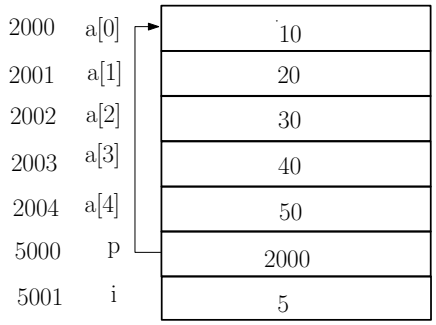

```
#include<stdio.h>
int main()
{
  int a[10], b, *p;
  a[0]=10;
  a[1]=20;
  p=a;
  *p = *p + 1;
  printf("a[0]=%d a[1]=%d *p=%d\n",a[0],a[1],*p);
  p=p+1;
  *p = *p + 1;
  printf("a[0]=%d a[1]=%d *p=%d\n",a[0],a[1],*p);
  *(p-1)=30;
→  printf("a[0]=%d a[1]=%d *p=%d\n",a[0],a[1],*p);
  return(0);
}
```

After executing last *printf*, output `a[0]=30 a[1]=21 *p=21` is displayed on screen.

The following program is another example of simultaneous usage of pointers and arrays.

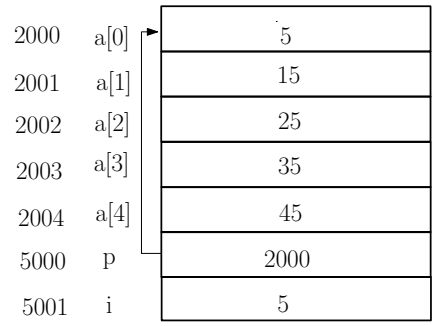
```
int main()
{
   int a[5], *p, i;
   i=0;
   while(i<5)
   {
       *(a+i)=(i+1)*10;
       i=i+1;
   }
   i=0;
   while(i<5)
   {
       printf("a[%d]=%d\n",i, a[i]);
       i=i+1;
   }
   p=a;
   i=0;
   while(i<5)
   {
       printf("*(p+%d)=%d *(a+%d)=%d\n",i, *(p+i), i, *(a+i) );
       i=i+1;
   }
   i=0;
   while(i<5)
   {
       *(p+i)=*(p+i)-5;
       i=i+1;
   }
   i=0;
   while(i<5)
   {
       printf("a[%d]=%d p[%d]=%d \n",i, a[i],i,p[i]);
       i=i+1;
   }
   return(0);
}
```

The memory state diagram of the above program after executing instructions upto `p=a;` is shown below.




```
#include<stdio.h>
int main()
{
  int a[5], *p, i;
  i=0;
  while(i<5)
  {
    *(a+i)=(i+1)*10;
    i=i+1;
  }
  i=0;
  while(i<5)
  {
    printf("a[%d]=%d\n",i, a[i]);
    i=i+1;
  }
  p=a;
→ i=0;
  while(i<5)
  {
    printf("*(p+%d)=%d *(a+%d)=%d\n",i, *(p+i), i, *(a+i) );
    i=i+1;
  }
  i=0;
  while(i<5)
  {
    *(p+i)=*(p+i)-5;
    i=i+1;
  }
  i=0;
  while(i<5)
  {
    printf("a[%d]=%d p[%d]=%d \n",i, a[i],i,p[i]);
    i=i+1;
  }
  return(0);
}
```

The memory state diagram after executing the fourth *while* loop is given below.




```
#include<stdio.h>
int main()
{
  int a[5], *p, i;
  i=0;
  while(i<5)
  {
    *(a+i)=(i+1)*10;
    i=i+1;
  }
  i=0;
  while(i<5)
  {
    printf("a[%d]=%d\n",i, a[i]);
    i=i+1;
  }
  p=a;
  i=0;
  while(i<5)
  {
    printf("*(p+%d)=%d *(a+%d)=%d\n",i, *(p+i), i, *(a+i) );
    i=i+1;
  }
  i=0;
  while(i<5)
  {
    *(p+i)=*(p+i)-5;
    i=i+1;
  }
  i=0;
  while(i<5)
  {
    printf("a[%d]=%d p[%d]=%d \n",i, a[i],i,p[i]);
    i=i+1;
  }
  return(0);
}
```

The output of the above program is as follows.

```
a[0]=10
a[1]=20
a[2]=30
a[3]=40
a[4]=50
*(p+0)=10 *(a+0)=10
*(p+1)=20 *(a+1)=20
*(p+2)=30 *(a+2)=30
*(p+3)=40 *(a+3)=40
*(p+4)=50 *(a+4)=50
a[0]=5 p[0]=5
a[1]=15 p[1]=15
a[2]=25 p[2]=25
a[3]=35 p[3]=35
a[4]=45 p[4]=45
```