Lecture 6
Pointers-Part 1

Dr. Hacer Yalım Keleş

Ref: Programming in ANSI C, Kumar

Pointers

pointers, the most sophisticated feature of C.

they lead to more efficient and compact code.

In particular, pointers enable us to

- achieve parameter passing by reference,
- deal concisely and effectively with arrays,
- represent complex data structures,
- work with dynamically allocated memory.

BASICS OF POINTERS

Memory can be visualized as an ordered sequence of consecutively numbered storage locations.

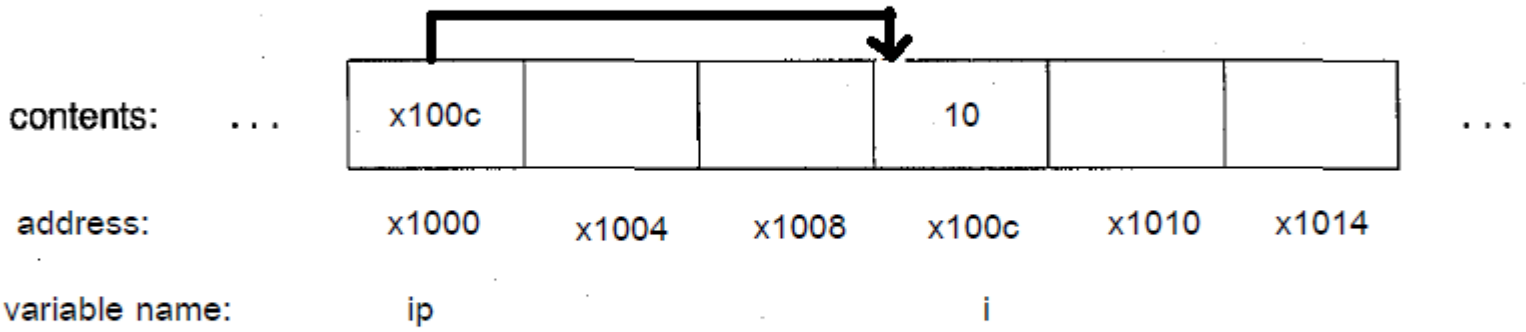
A data item is stored in memory in one or more adjacent storage locations depending upon its type.

The address of a data item is the address of its first storage location.

This address can be stored in another data item and manipulated in a program.

The address of a data item is called a *pointer* to the data item, and a variable that holds an address is called a pointer variable.

if `ip` is a pointer variable that contains the address of `i`, an `int`, this situation can be depicted as shown below:



&i	i	ip	*ip
x100c	10	x100c	10

Address and Dereferencing Operators

C provides two unary operators, & and *,
for manipulating data using pointers.

The operator &, when applied to a variable, yields its address (pointer to the variable),

The operator *, when applied to an address (pointer), fetches the value at that address.

Address and Dereferencing Operators

For example, if the integer variable `i` has been allocated the storage location numbered xlOOc and contains the integer value 10, then the address of `i`, indicated as `&i`, is 0x100c, and the value at the address &i, indicated as `* (&i)`, is 10.

Address and Dereferencing Operators

Accessing an object through a pointer is called dereferencing,

and the operator * is referred to as the *dereferencing* or *indirection* operator.

The operator & is referred to as the *address* operator.

Address and Dereferencing Operators

The & operator can only be applied to an lvalue, and constructs like

```
&10          &'C'          & (x+3)
```

that involve the addresses of constants and expressions are not valid.

If the type of the operand is *T* then the type of the result is "pointer to *T*".

For example, if `i` is an `int`, then `&i` is of type "pointer to `int`".

The * operator can only be applied to a pointer. If the operand is "pointer to T", then the type of the result is *T*.

For example, if `ip` is a pointer to an integer, then the type of `* ip` is `int`.

The expression `*ip`, where `ip` is a pointer to integer `i`, can occur in any expression in any context where `i` can. Thus,

```
j = *ip + 10;
```

is equivalent to

```
j = i + 10;
```

and

```
k = ++(*ip);
```

is equivalent to

```
k = ++i;
```

Pointer Type Declaration

For each type of object that can be declared in C, a corresponding type of pointer can be declared.

type * *identifier;*

declares the *identifier* to be of type "pointer to *type*". Note that the declaration allocates space for the named pointer variable, but not for what it points to.

For example, to declare `cp` to be a pointer to an object of type `char`, `ip` a pointer to an object of type `int`, and `dp` a pointer to an object of type `double`, we write

```
char   *cp;
int    *ip;
double *dp;
```

Pointer Assignment

A pointer value may be assigned to another pointer of the same type. For example, in the program fragment

```
int i = 1, j, *ip;
ip = &i;
j = *ip;
*ip = 0;
```

the first assignment statement assigns the address of variable `i` to `ip`, the second assigns the value at address `ip`, that is, `1` to `j`, and finally the third assigns 0 to `i` since `*ip` is the same as `i`.

Note that the two statements

```
ip = &i;
j = *ip;
```

are equivalent to the single assignment

```
j = *(&i);
```

or to the assignment

```
j = i;
```

That is. the address operator & is the inverse of the dereferencing operator *.

Pointer Initialization

*type *identifier = initializer;*

The initializer must either evaluate to an address of previously defined data of appropriate type or it can be the `NULL` pointer. For example, the declaration

```
#include <stdio.h>
float *fp = NULL;
```

initializes `f p` to `NULL;` the declarations

```
short s;
short *sp = &s;
```

initialize `sp` to the address of `s;` and the declarations

```
char c[10];
char *cp = &c[4];
```

initialize `cp` to the address of the fifth element of the array `c.`

```c
#include <stdio.h>

int main(void)
  {
    int i, j = 1;
    int *jp1, *jp2 = &j;   /* jp2 points to j */

    jp1 = jp2;             /* jp1 also points to j */
    i = *jp1;              /* i gets the value of j */
    *jp2 = *jp1 + i;       /* i is added to j */

    printf ("i = %d, j = %d, *jp1 = %d, *jp2 = %d\n",
             i, j, *jp1, *jp2);
    return 0;
  }
```

This program prints:

```
i = 1, j = 2, *jpl = 2, * jp2 = 2
```