Basis and Practice in Programming

Chapter 10: Pointers

Prof. Tamer ABUHMED
College of Software

성균관대학교
SUNG KYUN KWAN UNIVERSITY(SKKU)

What do we have learned so far?

Input/Output

- Scanf (), printf(), getchar(), putchar(),gets(), puts(),

Variables & Datatypes

- int, float, double, long, char, short

Constant, String, Boolean

Loops (while, for, do while), branching (if else, switch, goto)

Functions

Arrays (one & two dimensions)

Structures

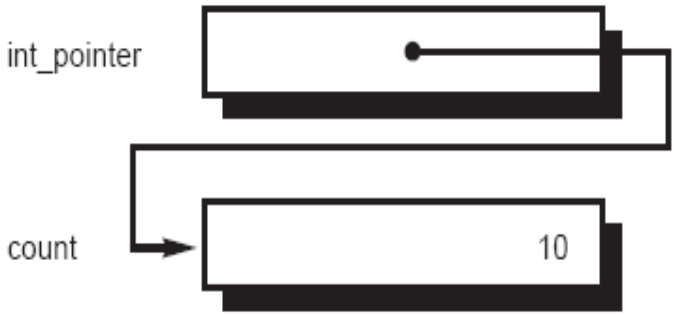
More are coming …

Lecture Objectives

- Introduction to Pointers
- Pointer operations
- Pointer Arithmetic
- Generic Pointers
- Arrays and Pointers
- More on Pointer Arithmetic

Pointers and addresses

- a ***pointer*** is a *variable* whose *value* is a *memory address*
- `int count = 10;`
- `int *int_pointer;`
- `int_pointer = &count;`
- The **address operator** has the effect of assigning to the variable `int_pointer`, not the value of `count`, but a *pointer* to the variable `count`.
- We say that `int_ptr` "points to" `count`
- The values and the format of the numbers representing memory addresses depend on the computer architecture and operating system. In order to have a portable way of representing memory addresses, we need a different type than integer !
- To print addresses: `%p`



Declaring pointer variables

```
type *  variable_name;
```

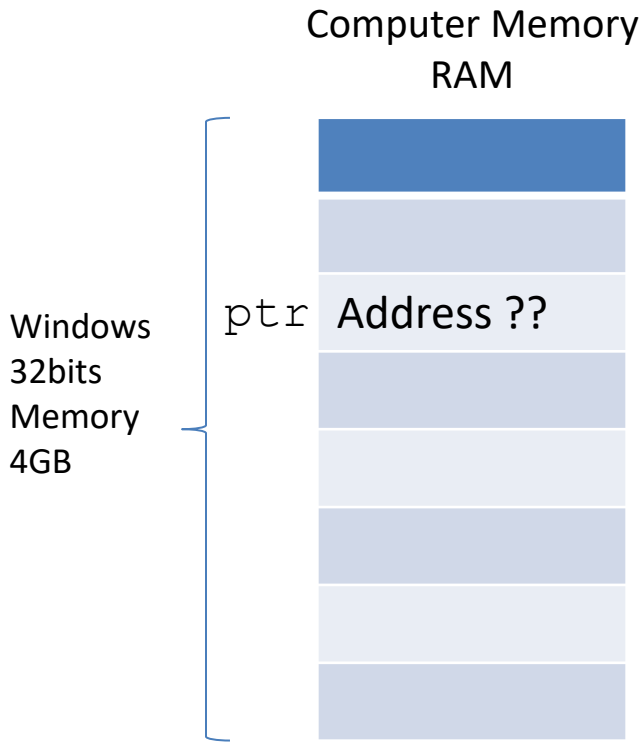
- it is not enough to say that a variable is a pointer. You also have to specify the *type of variable to which the pointer points !*
  - `int * p1; // p1 points to an integer`
  - `float *  p2; // p2 points to a float`
- Exception: generic pointers `(void *)` indicate that the pointed data type is unknown
  - may be used with explicit type cast to any type (type *)
  - `void * p;`

Pointers

- Special case of bounded-size natural numbers
  - Maximum memory limited by processor word-size
  - 2^{32} bytes = 4GB, 2^{64} bytes = 16 exabytes
- A pointer is just another kind of value
  - A basic type in C

```
int *ptr;
```

The variable “ptr” stores a pointer to an “int”.



Pointer Operations in C

- Creation

  & *variable* Returns variable's memory address

- Dereference

  * *pointer* Returns contents stored at address

- Indirect assignment

  * *pointer* = *val* Stores value at address

- Of course, still have...

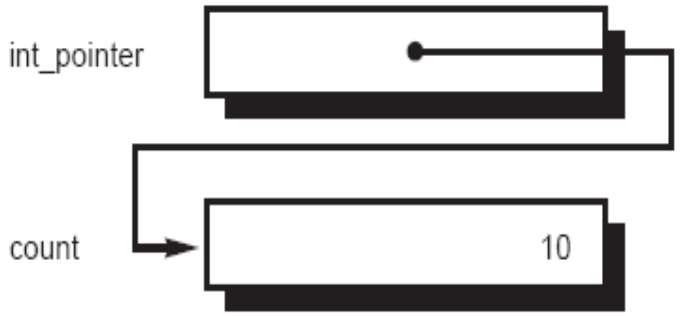
- Assignment

  pointer = *ptr* Stores pointer in another variable

Indirection (dereferencing) operator *

- To reference the contents of count through the pointer variable int_pointer, you use the ***indirection* operator**, which is the asterisk * as an unary prefix operator.
  `*int_pointer`
- If a pointer variable `p` has the type `t*`, then the expression `*p` has the type `t`

```
// Program to illustrate pointers
#include <stdio.h>
int main (void)
{
    int count = 10, x;
    int *int_pointer;
    int_pointer = &count;
    x = *int_pointer; //dereferencing
    printf ("count = %i, x = %i\n", count, x);
    return 0;
}
```



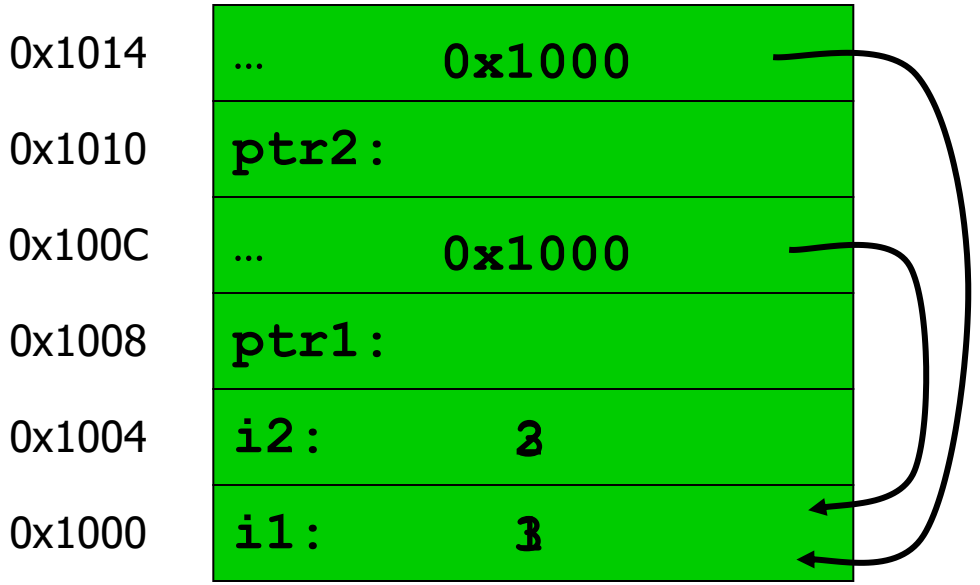
Using Pointers

```
int  i1;
int  i2;
int *ptr1;
int *ptr2;

i1 = 1;
i2 = 2;

ptr1 = &i1;
ptr2 = ptr1;

*ptr1 = 3;
i2 = *ptr2;
```



Using pointer variables

- The value of a pointer in C is meaningless until it is set pointing to something !
- How to set pointer values:
  - Using the address operator

```
int *p;
*p = 4;
```

Severe runtime error !!! the value 4 is stored in the location to which p points. But p, being uninitialized, has a random value, so we cannot know where the 4 will be stored !

  - Using directly assignements between pointer variables

```
int *p;
int  x;
p = &x;
*p = 4;
```

```
int *p;
int *p1;
int  x;
p1 = &x;
p = p1;
*p = 4;
```

Using Pointers (cont.)

```
int  int1     = 1036;   /* some data to point to  */
int  int2     = 8;

int *int_ptr1 = &int1;  /* get addresses of data  */
int *int_ptr2 = &int2;

*int_ptr1 = int_ptr2;

*int_ptr1 = int2;
```

What happens?

Type check warning: `int_ptr2` **is not an** `int`

`int1` **becomes 8**

Using Pointers (cont.)

```
int  int1     = 1036;   /* some data to point to  */
int  int2     = 8;

int *int_ptr1 = &int1;  /* get addresses of data  */
int *int_ptr2 = &int2;

int_ptr1 = *int_ptr2;

int_ptr1 = int_ptr2;
```

What happens?

Type check warning: `*int_ptr2` **is not an** `int *`

Changes `int_ptr1` **– doesn't change** `int1`

Pointer Arithmetic

pointer + number *pointer – number*

E.g., *pointer* + 1 adds 1 something to a pointer

```
char   *p;
char    a;
char    b;

p = &a;
p += 1;
```

In each, p now points to b
(Assuming compiler doesn't reorder variables in memory)

```
int   *p;
int    a;
int    b;

p = &a;
p += 1;
```

Adds 1*sizeof(char) to the memory address

Adds 1*sizeof(int) to the memory address

Pointer arithmetic should be used cautiously

const and pointers

- **With pointers, there are two things to consider:**
  - **whether the pointer will be changed**
  - **whether the value that the pointer points to will be changed.**
- Assume the following declarations:

```
char c = 'X';
char *charPtr = &c;
```

- If the pointer variable is always set pointing to c, it can be declared as a const pointer as follows:

```
char c, d;
char * const charPtr = &c;
*charPtr = 'Y'; // this is valid
charPtr = &d;   // not valid !!!
```

No address change

- If the location pointed to by charPtr will not change through the pointer variable charPtr, that can be noted with a declaration as follows:

```
const char *charPtr = &c;
charPtr = &d;   // this is valid
*charPtr = 'Y'; // not valid !!!
```

No data change

A Special Pointer in C

- Special constant pointer `NULL`
  - Points to no data
  - Dereferencing illegal – causes *segmentation fault*
  - To define, include `<stdlib.h>` or `<stdio.h>`
  - `Example:`

```
#include <stdio.h>
int main()
{

    int* ip = NULL;

    if (ip != NULL)  printf("value ip != NULL  %d\n", *ip);

    else if (ip == NULL)  printf("value ip = NULL %p\n", ip);

    return 0;
}
```

Generic Pointers

- void *: a "pointer to anything"

```
void   *p;
int     i;
char    c;
p = &i;
p = &c;
putchar(*(char *)p);
```

type cast: tells the compiler to "change" an object's type (for type checking purposes – does not modify the object in any way)

Dangerous! Sometimes necessary…

- Lose all information about what type of thing is pointed to
  - Reduces effectiveness of compiler's type-checking
  - Can't use pointer arithmetic

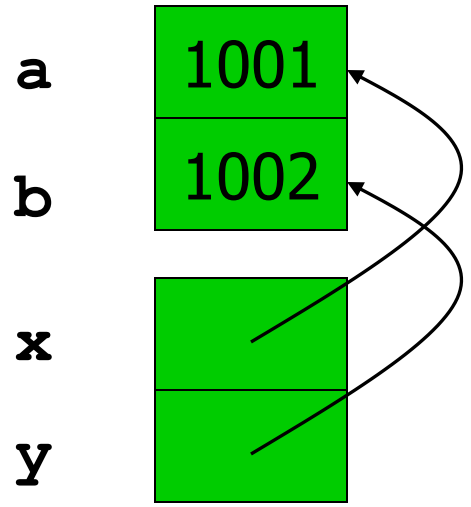
Pass-by-Reference

```
void set_x_and_y(int *x, int *y)
{

    *x = 1001;
    *y = 1002;
}

void f(void)
{
    int a = 1;
    int b = 2;

    set_x_and_y(&a, &b);
}
```



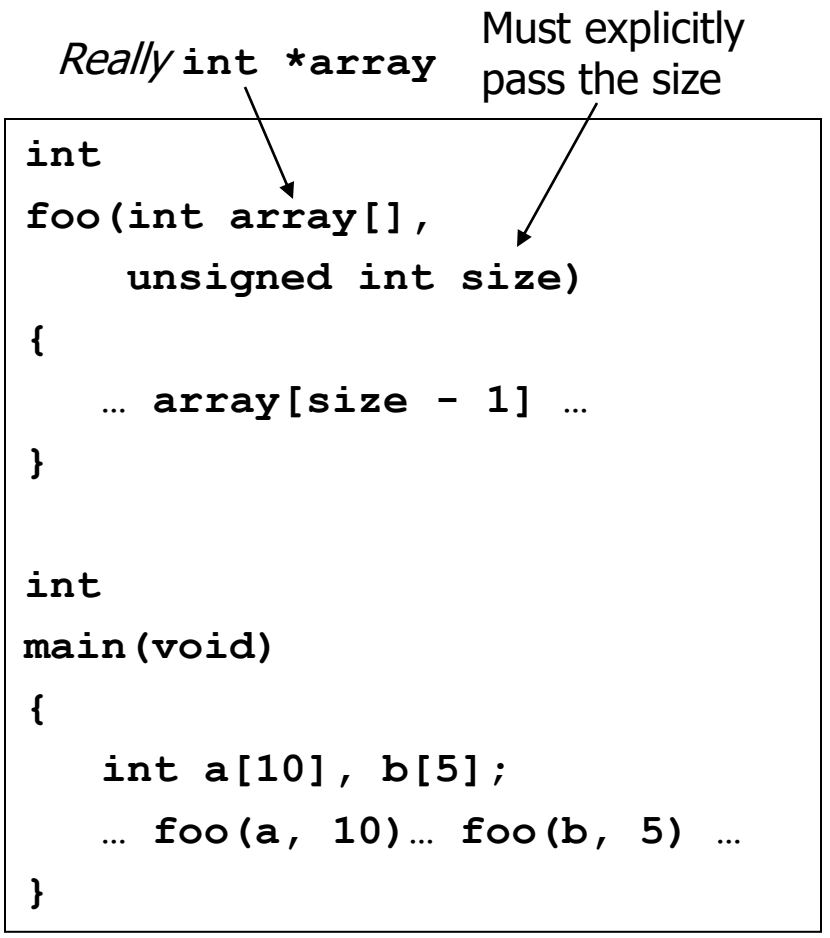
Arrays and Pointers

- Dirty "secret":
- Array name ≈ a pointer to the initial (0th) array element

```
a[i]  ≡  *(a + i)
```

- An array is passed to a function as a pointer
  - The array size is lost!
- Usually bad style to interchange arrays and pointers
  - Avoid pointer arithmetic!

Passing arrays:



Arrays and Pointers

```
int foo(int array[],
    unsigned int size)
{
   …
   printf("%d\n", sizeof(array));
}

int main(void)
{
   int a[10], b[5];
   … foo(a, 10)… foo(b, 5) …
   printf("%d\n", sizeof(a));
}
```

What does this print? **8**

... because **`array`** is really a pointer

What does this print? **40**

Arrays are **constant** pointers

```
int a[10];
int *pa;

pa=a;

pa++;
```

OK. Pointers are variables that can be assigned or incremented

```
int a[10];
int *pa;

a=pa;

a++;
```

Errors !!!

The name of an array is a CONSTANT having as a value the location of the first element.
You cannot change the address where the array is stored !
An array's name is equivalent with a ***constant*** pointer

Arrays and Pointers

```
int  i;
int  array[10];

for (i = 0; i < 10; i++)
{
  array[i] = …;
}
```

```
int *p;
int  array[10];

for (p = array; p < &array[10]; p++)
{
  *p = …;
}
```

These two blocks of code are functionally equivalent

Example: Arrays as parameters

```
void print1(int tab[], int N) {
    int i;
    for (i=0; i<N; i++)
        printf("%d ",tab[i]);
}
void print2(int tab[],int N) {
    int * ptr;
    for (ptr=tab; ptr<tab+N; ptr++)
        printf("%d ", *ptr);
}
void print3(int *tab,int N) {
    int * ptr;
    for (ptr=tab; ptr<tab+N; ptr++)
        printf("%d ", *ptr);
}
void print4(int *tab,int N) {
    int i;
    for (i=0; i<N; i++, tab++)
        printf("%d ", *tab);
}
```

The formal parameter can be declared as array or pointer !
In the body of the function, the array elements can be accessed through indexes or pointers !

```
void main(void) {
    int a[5]={1,2,3,4,5};
    print1(a,5);
    print2(a,5);
    print3(a,5);
    print4(a,5);
}
```

Pointer arithmetic

- **Increment/decrement:** if *p is a pointer to type T, p++ increases the value of p* by *sizeof(T) (sizeof(T) is the amount of storage needed for an object of type T).* Similarly, *p--* decreases *p* by *sizeof(T);*

```
T tab[N];
T * p;
int i;
p=&tab[i];
p++;   // p contains the address of tab[i+1];
```

- **Addition/subtraction with an integer**: if *p is a pointer to type T* and *n* an integer, *p+n increases the value of p* by *n*sizeof(T).* Similarly, *p-n* decreases *p* by *n*sizeof(T);*

```
T tab[N];
T * p;
p=tab;
p=p+n;   // p contains the address of tab[n].
```

Pointer arithmetic

- **Comparison of two pointers.**
- ***If p and q point to members of the same array***, then relations like ==, !=, <, >=, etc., work properly.
  - For example, p < q is true if p points to an earlier element of the array than q does.
- Any pointer can be meaningfully compared for equality or inequality with zero.
- **Pointer subtraction** :
- ***if p and q point to elements of the same array,*** and p<q, then q-p+1 is the number of elements from p to q inclusive.

- The behavior is undefined for arithmetic or comparisons with pointers that do not point to members of the same array.

Example

```
#include <stdio.h>
#include<stdbool.h>
bool compare_arrays(int* arry1, int size1, int* arry2, int size2)  {
      if (size1 != size2) {
            return false;
        }
        for (int i = 0; i < size1; i++)  {
              if (arry1+i == arry2+i)  {  // same array
                return true;
            }
              else if (*(arry1+i) != *(arry2+i))  { // not same array values
                  return false;

              }
        }
        return true;
}

int main(void)
{
    int size1 = 3, size2 = 3;
    int arr_one[] = { 1, 2, 3 };
    int arr_two[] = { 1, 2, 3 };
    int arr_three[] = { 1, 5, 3 };
    printf("%d",compare_arrays(arr_one, size1, arr_one, size2));
    printf("%d", compare_arrays(arr_one, size1, arr_two, size2));
    printf("%d", compare_arrays(arr_one, size1, arr_three, size2));
    return 0;
}
```

Lecture Summary

- Introduction to Pointers
- Pointer operations
- Pointer Arithmetic
- Generic Pointers
- Arrays and Pointers
- More on Pointer Arithmetic