Pointers II

Arash Rafiey

October 5, 2017

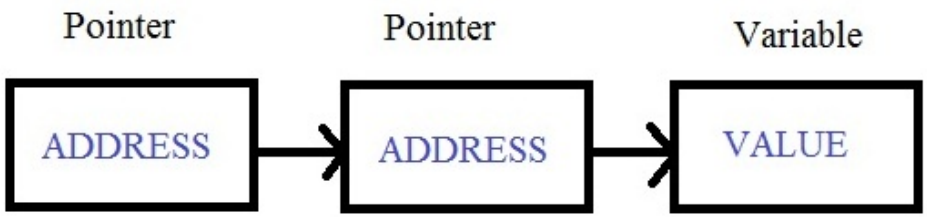
Pointer to pointer

A **pointer to a pointer** is a chain of pointers.

Pointer to pointer

A **pointer to a pointer** is a chain of pointers.

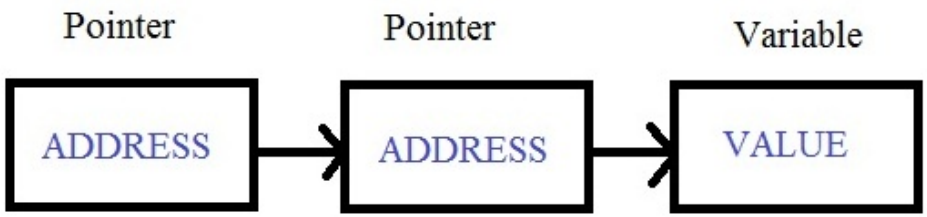
Pointer contains the address of a variable.



Pointer to pointer

A **pointer to a pointer** is a chain of pointers.

Pointer contains the address of a variable.

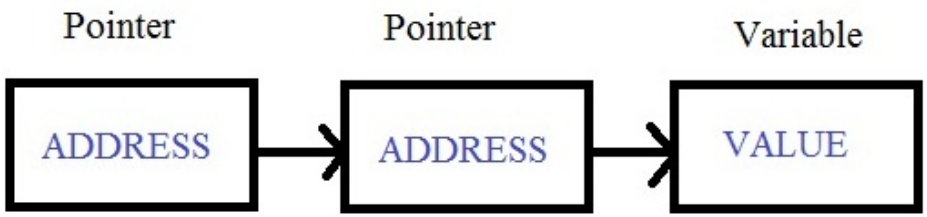


When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value of a variable.

Pointer to pointer

A **pointer to a pointer** is a chain of pointers.

Pointer contains the address of a variable.



When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value of a variable.

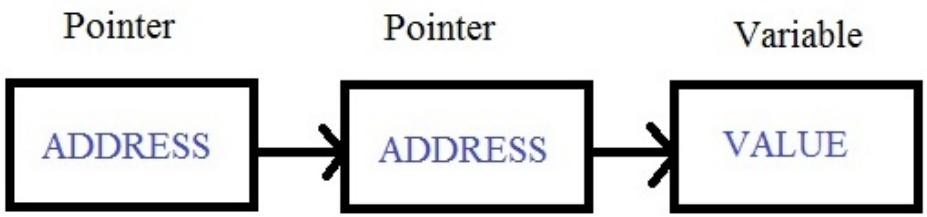
Declaration:

type ** pointer_name;

Pointer to pointer

A **pointer to a pointer** is a chain of pointers.

Pointer contains the address of a variable.



When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value of a variable.

Declaration:

type ** pointer_name;

Example:

```
int x = 0;
int *ptr;
int **pptr;
```

Pointer to pointer

```
ptr = &x; //points to the address of x
pptr = &ptr // points to the address of ptr
```

Pointer to pointer

ptr = &x; //points to the address of x

pptr = &ptr // points to the address of ptr

Here the address of pptr variable will have type of int ***.

Pointer to pointer

ptr = &x; //points to the address of x

pptr = &ptr // points to the address of ptr

Here the address of pptr variable will have type of int ***.

Consider an array has many lines of text.

Each line can be accessed by a pointer to its first character and the pointers can be stored in an array.

Pointer to pointer

ptr = &x; //points to the address of x

pptr = &ptr // points to the address of ptr

Here the address of pptr variable will have type of int ***.

Consider an array has many lines of text.

Each line can be accessed by a pointer to its first character and the pointers can be stored in an array.

To swap 2 lines, the **pointers are exchanged**, not the lines themselves.

This eliminates complicated storage management and high overhead associated with moving the lines.

Initialization of pointer arrays

Consider a function month_name(n) returns a pointer to a character string containing the name of the n-th month.

Initialization of pointer arrays

Consider a function month_name(n) returns a pointer to a character string containing the name of the n-th month.

month_name contains an array of character strings, and returns a pointer to the proper string when called.

Initialization of pointer arrays

Consider a function month_name(n) returns a pointer to a character string containing the name of the n-th month.

month_name contains an array of character strings, and returns a pointer to the proper string when called.

```
char *month_name(int n) {
  static char *name[ ] = {
   "Illegal month",
   "January", "February", "March",
   "April", "May", "June",
   "July", "August", "September",
   "October", "November", "December"
   }; //pointer to each string is stored in name[ ]
```

Initialization of pointer arrays

```
if ( n < 1 || n > 12 ) retrun name[0];
else
return name[n];
}
```

Initialization of pointer arrays

if ($n < 1$ || $n > 12$) retrun name[0];
else
return name[n];

}
name[ ] is an array of character pointers.

Initialization of pointer arrays

if ($n < 1$ || $n > 12$) retrun name[0];
else
return name[n];

}

name[ ] is an array of character pointers.

Each character string is assigned to a position in the memory.

Initialization of pointer arrays

if ($n < 1$ || $n > 12$) retrun name[0];
else
return name[n];

}

name[ ] is an array of character pointers.

Each character string is assigned to a position in the memory.

A pointer to each string is stored in the pointer array: name[ ].

Each pointer points to the start position of the string.

Initialization of pointer arrays

if ($n < 1$ || $n > 12$) retrun name[0];
else
return name[n];

}

name[ ] is an array of character pointers.

Each character string is assigned to a position in the memory.

A pointer to each string is stored in the pointer array: name[ ].

Each pointer points to the start position of the string.

The i-th string can be can be accessed through name[i]

Multi-dimensional arrays

Multi-dimensional array declaration:

type arrayName [d1][d2]...[dN];

Multi-dimensional arrays

Multi-dimensional array declaration:

type arrayName [d1][d2]...[dN];

//These two definitions of two-dimensional array are the same:

```
int costs[2][3] = {1, 2, 3, 4, 5, 6};
int costs[2][3] = { {1, 2, 3,} , //Row 0.
                    {4, 5, 6}, }; // Row 1.
```

Multi-dimensional arrays

Multi-dimensional array declaration:

type arrayName [d1][d2]...[dN];

//These two definitions of two-dimensional array are the same:

int costs[2][3] = {1, 2, 3, 4, 5, 6};

```
int costs[2][3] = { {1, 2, 3,} , //Row 0.
                    {4, 5, 6}, }; // Row 1.
```

To access the array: int value = costs[0][2]; **//value = 3**

This is not legal: costs[0, 1].

Three dimensional array: int seconds[24][60][60];

Day of year example

Convert the month and day into the day of the year.

Day of year example

Convert the month and day into the day of the year.

For example:

February 29 is converted to 60

//i.e february 29th is the 60th day of the year.

Day of year example

Convert the month and day into the day of the year.

For example:

February 29 is converted to 60

//i.e february 29th is the 60th day of the year.

Since the number of days differ for non-leap and leap year, we use two rows of a two dimensional array.

Day of year example

Convert the month and day into the day of the year.

For example:

February 29 is converted to 60

//i.e february 29th is the 60th day of the year.

Since the number of days differ for non-leap and leap year, we use two rows of a two dimensional array.

```
static char dayTab[2][13] = {
    //Non leap year.
    {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 },
    //Leap year.
    {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }
};
```

Day of year example continued

```
//set day of year from month and day
int day_of_year( int year, int month, int day )
{
   int i, leap;
   leap = (year%4 == 0) && (year%100 != 0) || (0 ==
year%400);
   //leap is either 0 or 1
   for( i = 1; i < month; i++ );
     day += dayTab [ leap ] [ i ];
   return day;
}
```

Pointers vs. Multi-dimensional arrays

Difference between a two-dimensional array and an array of pointers.

Pointers vs. Multi-dimensional arrays

Difference between a two-dimensional array and an array of pointers.

```
int a[10][20];
int *b[10];
```

Pointers vs. Multi-dimensional arrays

Difference between a two-dimensional array and an array of pointers.

```
int a[10][20];
int *b[10];
```

- a is a two-dimensional array with 200 int-sized locations set aside.

Pointers vs. Multi-dimensional arrays

Difference between a two-dimensional array and an array of pointers.

```
int a[10][20];
int *b[10];
```

- a is a two-dimensional array with 200 int-sized locations set aside.
- For b, 10 pointers are allocated. Initialization of these pointers must be done explicitly.

Pointers vs. Multi-dimensional arrays

Difference between a two-dimensional array and an array of pointers.

```
int a[10][20];
int *b[10];
```

- a is a two-dimensional array with 200 int-sized locations set aside.
- For b, 10 pointers are allocated. Initialization of these pointers must be done explicitly.
- Each element of b can be of different length.

Dynamic memory allocation

User specifies the file size to create at **run-time** - it is not possible to know it in advance.

Dynamic memory allocation

User specifies the file size to create at **run-time** - it is not possible to know it in advance.

1. If file size is small, program may preallocate a very large chunk of memory.
2. Otherwise, big file won't fit in memory.

Dynamic memory allocation

User specifies the file size to create at **run-time** - it is not possible to know it in advance.

1. If file size is small, program may preallocate a very large chunk of memory.
2. Otherwise, big file won't fit in memory.

Possible solution: allocate memory **dynamically** when it is needed.

Dynamic memory allocation

User specifies the file size to create at **run-time** - it is not possible to know it in advance.

1. If file size is small, program may preallocate a very large chunk of memory.
2. Otherwise, big file won't fit in memory.

Possible solution: allocate memory **dynamically** when it is needed.

C can allocate memory **statically** (e.g. global variables once when program starts), **dynamically** (malloc, on heap), **automatically** (when function is executed, on stack).

Dynamic memory allocation

User specifies the file size to create at **run-time** - it is not possible to know it in advance.

1. If file size is small, program may preallocate a very large chunk of memory.
2. Otherwise, big file won't fit in memory.

Possible solution: allocate memory **dynamically** when it is needed.

C can allocate memory **statically** (e.g. global variables once when program starts), **dynamically** (malloc, on heap), **automatically** (when function is executed, on stack).

When memory block is no longer needed, it should be released to return it to the **Operating System**.

Dynamic memory allocation

User specifies the file size to create at **run-time** - it is not possible to know it in advance.

1. If file size is small, program may preallocate a very large chunk of memory.
2. Otherwise, big file won't fit in memory.

Possible solution: allocate memory **dynamically** when it is needed.

C can allocate memory **statically** (e.g. global variables once when program starts), **dynamically** (malloc, on heap), **automatically** (when function is executed, on stack).

When memory block is no longer needed, it should be released to return it to the **Operating System**.

Function **signature** to free the memory, previously allocated by malloc, calloc, realloc:
void **free** (void* ptr);

Dynamic memory allocation functions

When array is declared as int[10], it is **allocated on the stack**.
When function returns, that memory is "released" **automatically**.

Dynamic memory allocation functions

When array is declared as int[10], it is **allocated on the stack**. When function returns, that memory is "released" **automatically**.

To allocate block of memory on **the heap**, C uses **malloc** functions.

Dynamic memory allocation functions

When array is declared as int[10], it is **allocated on the stack**. When function returns, that memory is "released" **automatically**.

To allocate block of memory on **the heap**, C uses **malloc** functions.

void* **malloc** (size_t size) is used to allocate block of "size" bytes of memory and return pointer to block beginning.

Dynamic memory allocation functions

When array is declared as int[10], it is **allocated on the stack**. When function returns, that memory is "released" **automatically**.

To allocate block of memory on **the heap**, C uses **malloc** functions.

void* **malloc** (size_t size) is used to allocate block of "size" bytes of memory and return pointer to block beginning.

1. The "size" is limited by the amount of available memory.
2. The allocated memory is uninitialized.
3. If function fails, NULL is returned.

Dynamic memory allocation functions

When array is declared as int[10], it is **allocated on the stack**. When function returns, that memory is "released" **automatically**.

To allocate block of memory on **the heap**, C uses **malloc** functions.

void* **malloc** (size_t size) is used to allocate block of "size" bytes of memory and return pointer to block beginning.

1. The "size" is limited by the amount of available memory.
2. The allocated memory is uninitialized.
3. If function fails, NULL is returned.

```
# include <stdio.h>
# include <stdlib.h>
int main() {
int i,j;
char *ptr;
ptr= malloc(15);
ptr="this is a test";
int *p=malloc(15);
p[0]=17;
printf("%s \n", ptr);
printf( "first element is %d \n ", p[0]);
// free(ptr); // don't do it because ptr points to a constant string
free(p);
}
```