Pointers and Arrays

Arash Rafiey

October 3, 2017

Memory and pointers

Operating System runs programs, assigns **physical addresses**.

Memory and pointers

Operating System runs programs, assigns **physical addresses**.

Memory is set of cells, that can be combined into groups.

Memory and pointers

Operating System runs programs, assigns **physical addresses**.

Memory is set of cells, that can be combined into groups.

Pointer is group of cells, holding address.

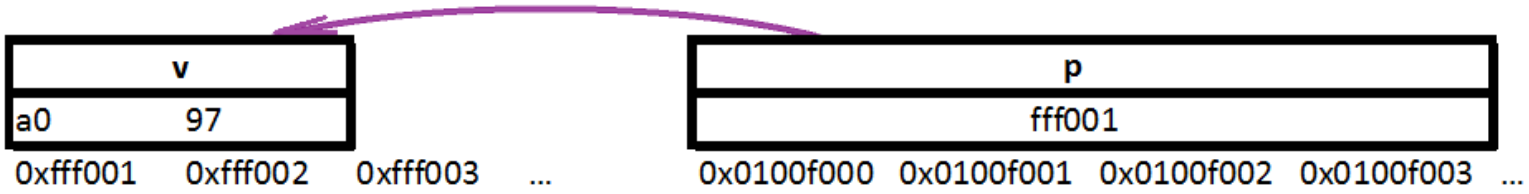
Memory and pointers

Operating System runs programs, assigns **physical addresses**.

Memory is set of cells, that can be combined into groups.

Pointer is group of cells, holding address.

Let variable v have type short; **pointer** p points to v:



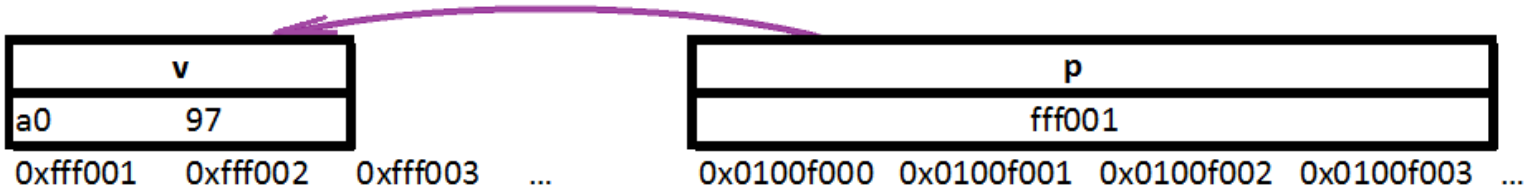
Memory and pointers

Operating System runs programs, assigns **physical addresses**.

Memory is set of cells, that can be combined into groups.

Pointer is group of cells, holding address.

Let variable v have type short; **pointer** p points to v:



8 memory cells (8 bytes) can hold:

1. one instance of long long int //64 bits.
2. one - double
3. two - int //2 * 32 = 64 bits.
4. four - short
5. eight - char

Variables and pointers

Variable has name (**identifier**) and **value** (in computer's memory).

Variables and pointers

Variable has name (**identifier**) and **value** (in computer's memory).

Abstraction: Using variables allows manipulating their values, ignoring where the variable will be stored physically.

Variables and pointers

Variable has name (**identifier**) and **value** (in computer's memory).

Abstraction: Using variables allows manipulating their values, ignoring where the variable will be stored physically.

Pointer is a variable storing the **address** of another variable.

Variables and pointers

Variable has name (**identifier**) and **value** (in computer's memory).

Abstraction: Using variables allows manipulating their values, ignoring where the variable will be stored physically.

Pointer is a variable storing the **address** of another variable.

Pointers are powerful tool like ... - be careful!

Variables and pointers

Variable has name (**identifier**) and **value** (in computer's memory).

Abstraction: Using variables allows manipulating their values, ignoring where the variable will be stored physically.

Pointer is a variable storing the **address** of another variable.

Pointers are powerful tool like ... - be careful!

Pointer declaration syntax uses **asterisk** *:

type * name;

Variables and pointers

Variable has name (**identifier**) and **value** (in computer's memory).

Abstraction: Using variables allows manipulating their values, ignoring where the variable will be stored physically.

Pointer is a variable storing the **address** of another variable.

Pointers are powerful tool like ... - be careful!

Pointer declaration syntax uses **asterisk** *:

type * name;

Pointer declaration examples:

1. int* count;
2. char* name;
3. double* ratio;
4. void* data; //**Pointer to void is generic pointer.**

Pointers and addresses

The **unary operator &** (ampersand) gets the address of a variable.

Pointers and addresses

The **unary operator &** (ampersand) gets the address of a variable.
The address is known at run-time only.

Pointers and addresses

The **unary operator &** (ampersand) gets the address of a variable.
The address is known at run-time only.
Dereferencing operator * returns the value the pointer points to.

Pointers and addresses

The **unary operator &** (ampersand) gets the address of a variable.
The address is known at run-time only.
Dereferencing operator * returns the value the pointer points to.

Example:

```
int days = 25;
int* address = &days; //Address now points to days
* address += 1; //The days value becomes 26.
int count = * address; //Value of count will be 26.
```

Pointers and addresses

The **unary operator &** (ampersand) gets the address of a variable.
The address is known at run-time only.
Dereferencing operator * returns the value the pointer points to.

Example:

```
int days = 25;
int* address = &days; //Address now points to days
* address += 1; //The days value becomes 26.
int count = * address; //Value of count will be 26.
```

The last line of the above example can be read as "**count** is equal to value pointed to by address".

Pointer dereference

Pointer points to specific data type except for void * which cannot be **dereferenced**.

Pointer dereference

Pointer points to specific data type except for void * which cannot be **dereferenced**.

Pointer dereference

Pointer points to specific data type except for void * which cannot be **dereferenced**.

Consider:

```
int days = 25;
int* p = &days;
```

Pointer dereference

Pointer points to specific data type except for void * which cannot be **dereferenced**.

Consider:
int days = 25;
int* p = &days;
To increment the value p points to:

Pointer dereference

Pointer points to specific data type except for void * which cannot be **dereferenced**.

Consider:

```
int days = 25;
int* p = &days;
```

To increment the value p points to:

1. `*p = *p + 1;`
2. `*p = 1 + *p;`
3. `*p += 1;`
4. `++*p;`
5. `++(*p); //This is a safer way.`
6. `(*p)++;`

Pointer dereference

Pointer points to specific data type except for void * which cannot be **dereferenced**.

Consider:

```
int days = 25;
int* p = &days;
```

To increment the value p points to:

1. `*p = *p + 1;`
2. `*p = 1 + *p;`
3. `*p += 1;`
4. `++*p;`
5. `++(*p); //This is a safer way.`
6. `(*p)++;`

Parentheses are mandatory in previous line because ***p++** will **increment p** by 1 instead of incrementing days. This is because dereference * associate **right-to-left** and **postfix increment** has highest **precedence**.

Quick review of C Operator Precedence

The following table lists the precedence and associativity of C operators. Operators are listed top to bottom, in descending precedence.

Precedence	Operator	Description	Associativity
1	++ --	Suffix/postfix increment and decrement	Left-to-right
	( )	Function call	
	[ ]	Array subscripting	
	.	Structure and union member access	
	->	Structure and union member access through pointer	
	(*type*){*list*}	Compound literal(C99)	
2	++ --	Prefix increment and decrement	Right-to-left
	+ -	Unary plus and minus	
	! ~	Logical NOT and bitwise NOT	
	(*type*)	Type cast	
	*	Indirection (dereference)	
	&	Address-of	
	sizeof	Size-of	
	_Alignof	Alignment requirement(C11)	
3	* / %	Multiplication, division, and remainder	Left-to-right
4	+ -	Addition and subtraction	
5	<< >>	Bitwise left shift and right shift	
6	< <=	For relational operators < and ≤ respectively	
	> >=	For relational operators > and ≥ respectively	
7	== !=	For relational = and ≠ respectively	
8	&	Bitwise AND	
9	^	Bitwise XOR (exclusive or)	

Constant pointers

Solution for previous slide example is to use **constant pointers**:

Constant pointers

Solution for previous slide example is to use **constant pointers**:

1. **Constant pointers** can change the value they point to.
2. But the pointers themselves cannot change (are const).
3. They are declared as *const.

Constant pointers

Solution for previous slide example is to use **constant pointers**:

1. **Constant pointers** can change the value they point to.
2. But the pointers themselves cannot change (are const).
3. They are declared as *const.

Example:

```
int days = 25;
int* const p = &days;
```

Constant pointers

Solution for previous slide example is to use **constant pointers**:

1. **Constant pointers** can change the value they point to.
2. But the pointers themselves cannot change (are const).
3. They are declared as *const.

Example:

```
int days = 25;
int* const p = &days;

++(*p); //days is 26.
*p = 27; //days is 27.
```

Constant pointers

Solution for previous slide example is to use **constant pointers**:

1. **Constant pointers** can change the value they point to.
2. But the pointers themselves cannot change (are const).
3. They are declared as *const.

Example:

```
int days = 25;
int* const p = &days;

++(*p); //days is 26.
*p = 27; //days is 27.

*(++p) = 28; //Compilation error.
*p++; //Cannot assign value to const variable p.
```

The two last lines of this example will not compile.

Pointers to constant

Constant pointers:

Pointers to constant

Constant pointers:

1. Can change the value they point to.
2. But the pointers themselves cannot change (are const).
3. They are declared as *const.

Pointers to constant

Constant pointers:

1. Can change the value they point to.
2. But the pointers themselves cannot change (are const).
3. They are declared as *const.

Pointers to constant:

Pointers to constant

Constant pointers:

1. Can change the value they point to.
2. But the pointers themselves cannot change (are const).
3. They are declared as *const.

Pointers to constant:

1. Cannot change the value they point to (value is const).
2. But the pointers can be changed.
3. They are declared as const*.

Pointers to constant

Constant pointers:

1. Can change the value they point to.
2. But the pointers themselves cannot change (are const).
3. They are declared as *const.

Pointers to constant:

1. Cannot change the value they point to (value is const).
2. But the pointers can be changed.
3. They are declared as const*.

Example:
```
int days = 25;
int month = 1;
const int* p = &days;
++p; //p would probably point to month - do not do this.
*p = 26; //Compilation error.
```

Pointers and function arguments

C passes function arguments **by value**.

Pointers and function arguments

C passes function arguments **by value**.

There is no direct way for a called function to alter a variable in the calling function.

Pointers and function arguments

C passes function arguments **by value**.

There is no direct way for a called function to alter a variable in the calling function.

Example:

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

Pointers and function arguments

C passes function arguments **by value**.

There is no direct way for a called function to alter a variable in the calling function.

Example:

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

swap(a, b) interchanges only copies of a and b, but does not change the actual arguments a and b.

Pointers and function arguments

C passes function arguments **by value**.

There is no direct way for a called function to alter a variable in the calling function.

Example:

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

swap(a, b) interchanges only copies of a and b, but does not change the actual arguments a and b.

Solution is to use **pointers**.

Pointers and function arguments

Example:

```
void swap(int *px, int *py);
{
int temp;
temp = *px;
*px = *py;
*py = temp;
}
```

Pointers and function arguments

Example:

```
void swap(int *px, int *py);
{
int temp;
temp = *px;
*px = *py;
*py = temp;
}
```

swap(&a, &b) sends the address of a and b as arguments.

So void swap(int *pa, int *pb) function interchanges the actual arguments a and b.

Swap via pointers

Example:

```
# include<stdio.h>
void swap( int *a, int *b);
void main() {
  int x = 10, y = 20;
  printf("Values before swap in main x = %d, y = %d \n",x,y);
  //x=10 y=20
  swap(&x,&y);
  printf("Values after swap in main x = %d, y = %d \n",x,y);
  //x=20 y=10
}
```

Swap via pointers

```
void swap( int *a, int *b )
{
  int temp;
  temp = *a;
  *a = *b;
  *b = temp;
  printf(" Values after swap inside swap function: x = %d, y = %d", *a, *b);
  //x=20 y=10
}
```

Pointers and arrays

Array is block of objects of **same type**, stored **consecutively** in memory.

Pointers and arrays

Array is block of objects of **same type**, stored **consecutively** in memory.

Instead of declaring many variables, such as a0, a1, ..., a9, declare one array variable: `int a[10];`

Pointers and arrays

Array is block of objects of **same type**, stored **consecutively** in memory.

Instead of declaring many variables, such as a0, a1, ..., a9, declare one array variable: `int a[10];`

n-th **element** in array is accessed by **index** n, that starts from 0.

Pointers and arrays

Array is block of objects of **same type**, stored **consecutively** in memory.

Instead of declaring many variables, such as a0, a1, ..., a9, declare one array variable: `int a[10];`

n-th **element** in array is accessed by **index** n, that starts from 0.

1. a[0] is zeroth element, n is zero.
2. a[n] is the n-th element.
3. a[9] is the last element, n is 9.

Pointers and arrays

Array is block of objects of **same type**, stored **consecutively** in memory.

Instead of declaring many variables, such as a0, a1, ..., a9, declare one array variable: `int a[10];`

n-th **element** in array is accessed by **index** n, that starts from 0.

1. a[0] is zeroth element, n is zero.
2. a[n] is the n-th element.
3. a[9] is the last element, n is 9.

```
int *pa = &a[0];
```

pa points to element zero of array a, or pa contains address of a[0].

Pointers and arrays

Array is block of objects of **same type**, stored **consecutively** in memory.

Instead of declaring many variables, such as a0, a1, ..., a9, declare one array variable: int a[10];

n-th **element** in array is accessed by **index** n, that starts from 0.

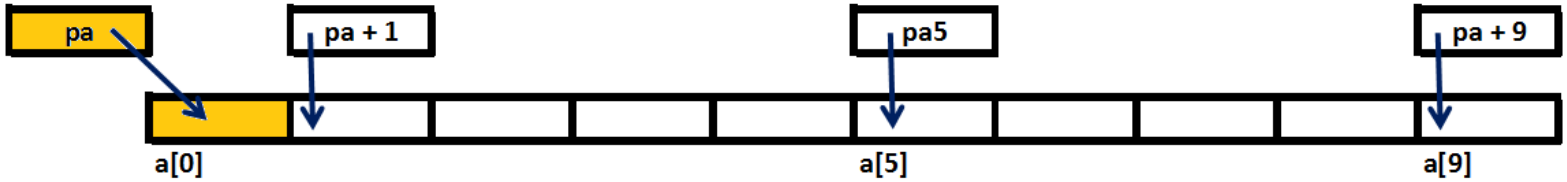
1. a[0] is zeroth element, n is zero.
2. a[n] is the n-th element.
3. a[9] is the last element, n is 9.

```
int *pa = &a[0];
```

pa points to element zero of array a, or pa contains address of a[0].

```
int b = *pa; //Copy the contents of a[0] into b.
```

Pointer arithmetic

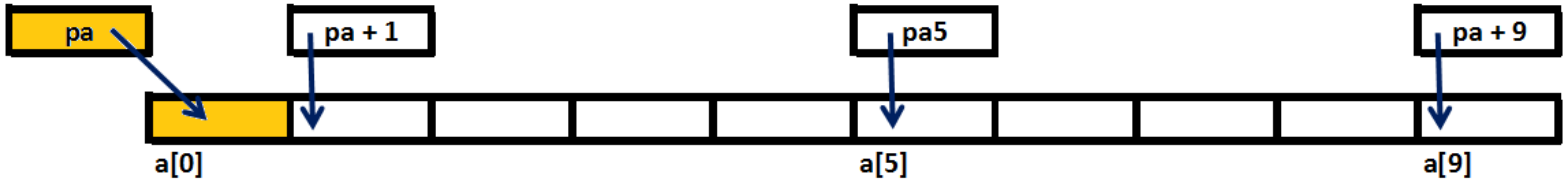



```
int a[10];

int *pa = &a[0];

int *pa5 = a + 5;
```

Pointer arithmetic




```
int a[10];
int *pa = &a[0];
int *pa5 = a + 5;
```

1. pa points to 0-th element of array a.
2. pa5 points to 5-th element of a.
3. (pa5 + 1) points to next (6-th) element.
4. (pa5 - n) points n elements before pa5.
5. *(pa+1) refers to contents of a[1].

Pointer arithmetic operations

Pointer manipulations **automatically consider size** of the type.

Pointer arithmetic operations

Pointer manipulations **automatically consider size** of the type.

Allowed pointer operations are:

Pointer arithmetic operations

Pointer manipulations **automatically consider size** of the type.

Allowed pointer operations are:

1. **Assignment** of pointers of same type.
   Example: int a[10];
   int *p1 = &a[9];
   int *p2;
   p2 = p1;//Both p1 and p2 will point to the same memory location.

Pointer arithmetic operations

Pointer manipulations **automatically consider size** of the type.

Allowed pointer operations are:

1. **Assignment** of pointers of same type.
   Example: int a[10];
   int *p1 = &a[9];
   int *p2;
   p2 = p1;//Both p1 and p2 will point to the same memory location.
2. **Subtraction and addition** of pointer and integer type.
   If p1 points to a particular element of an array , then p1+1 points to the next element.

Pointer arithmetic operations

Pointer manipulations **automatically consider size** of the type.

Allowed pointer operations are:

1. **Assignment** of pointers of same type.
   Example: int a[10];
   int *p1 = &a[9];
   int *p2;
   p2 = p1;//Both p1 and p2 will point to the same memory location.
2. **Subtraction and addition** of pointer and integer type.
   If p1 points to a particular element of an array , then p1+1 points to the next element.
3. **Subtraction or comparison** of two pointers to same array.
   Example: int a[10];
   int *p1 = a + 2;
   int *p2 = a + 5;
   printf("%d", p2-p1);// prints 3

Pointer arithmetic operations

Prohibited pointer operations:

Pointer arithmetic operations

Prohibited pointer operations:

1. Add/multiply/divide/shift/mask (+, *, /, << or >>, | or & or ^) two pointers .

Pointer arithmetic operations

Prohibited pointer operations:

1. Add/multiply/divide/shift/mask (+, *, /, << or >>, | or & or ^) two pointers .
2. Add/multiply/divide a float or double to pointer.

Pointer arithmetic operations

Prohibited pointer operations:

1. Add/multiply/divide/shift/mask (+, *, /, << or >>, | or & or ^) two pointers .
2. Add/multiply/divide a float or double to pointer.
3. Pointer of one type cannot be assigned to pointer of another type.

   Example:

```
int*p1;
char*b = p1; //Error
char*valid_cast = (char*)p1;
```

Array name

Pointer to array:

```
int a[10];
int *pa = &a[0];
```

Array name

Pointer to array:

```
int a[10];
int *pa = &a[0];
```

Difference between array name and pointer:

Array name

Pointer to array:

```
int a[10];
int *pa = &a[0];
```

Difference between array name and pointer:

Pointer is a variable. So

1. pa = a;
2. pa ++;

are legal.

Array name

Pointer to array:

```
int a[10];
int *pa = &a[0];
```

Difference between array name and pointer:

Pointer is a variable. So

1. pa = a;
2. pa ++;

are legal.

Array name is not a variable, but points to **initial element**. So

Array name

Pointer to array:

```
int a[10];
int *pa = &a[0];
```

Difference between array name and pointer:

Pointer is a variable. So

1. pa = a;
2. pa ++;

are legal.

Array name is not a variable, but points to **initial element**. So

1. a = pa;
2. ++ a;

are illegal.

Character pointers

Character strings are accessed through character pointers.

```
char *pmsg;
pmsg = "HELLO";
```

Character pointers

Character strings are accessed through character pointers.

```
char *pmsg;
pmsg = "HELLO";
```

Character pointer points to the first element.

This is **not string copying**. C does not provide operators for processing entire string of characters as unit.

Character pointers

Character strings are accessed through character pointers.

```
char *pmsg;
pmsg = "HELLO";
```

Character pointer points to the first element.

This is **not string copying**. C does not provide operators for processing entire string of characters as unit.

1. `char a1[] = "good news";`
2. `char *p2 = "good news";`

Character pointers

Character strings are accessed through character pointers.

```
char *pmsg;
pmsg = "HELLO";
```

Character pointer points to the first element.

This is **not string copying**. C does not provide operators for processing entire string of characters as unit.

1. `char a1[] = "good news";`
2. `char *p2 = "good news";`

What is the difference between the two?

Character pointers

Character strings are accessed through character pointers.

```
char *pmsg;
pmsg = "HELLO";
```

Character pointer points to the first element.

This is **not string copying**. C does not provide operators for processing entire string of characters as unit.

1. `char a1[] = "good news";`
2. `char *p2 = "good news";`

What is the difference between the two?

1. a1 is an **array** of 10 chars that always refers to same storage, although characters within array may be changed.
2. p2 is **pointer** to **string constant**. If this string constant is modified, result is undefined.

Character pointers and functions

The C library function **strcpy**(dest,src) copies the string pointed by **src** to **dest**.

Character pointers and functions

The C library function **strcpy**(dest,src) copies the string pointed by **src** to **dest**.

void *strcpy(char *dest, char *src)

Character pointers and functions

The C library function **strcpy**(dest,src) copies the string pointed by **src** to **dest**.

void *strcpy(char *dest, char *src)

strcpy using array:

```
void strcpy1(char *t, char const*s) {
   int i = 0;
   while ('\0' != (t[i] = s[i]))
     ++i; }
```

Character pointers and functions

The C library function **strcpy**(dest,src) copies the string pointed by **src** to **dest**.

void *strcpy(char *dest, char *src)

strcpy using array:

```
void strcpy1(char *t, char const*s) {
   int i = 0;
   while ('\0' != (t[i] = s[i]))
     ++i; }
```

strcpy using pointers:

```
void strcpy2(char *t, char const*s) {
   while ('\0' != (*t = *s)) {
     ++s;
     ++t; }
}
```

String is copied each character at a time, until '\0'.

strcmp - string compare

strcmp(s, t) compares the string pointed by s to the string pointed by t.

strcmp - string compare

strcmp(s, t) compares the string pointed by s to the string pointed by t.

It returns:

- **zero** if the strings are equal.
- **negative** if s is lexicographically less than t
- **positive** if s is lexicographically greater than t

strcmp - string compare

strcmp(s, t) compares the string pointed by s to the string pointed by t.

It returns:

- **zero** if the strings are equal.
- **negative** if s is lexicographically less than t
- **positive** if s is lexicographically greater than t

strcmp using array:

```
int strcmp1(char *s, char const*t) {
   int i;
   for(i=0; s[i] && t[i]; i++ )
     if(s[i] != t[i] )
        retun 0;
   if( s[i] == '\0' && t[i] ) return 0;
   else return 1;
}
```