

C vs. Python Syntax

Spring 2017 – Version 5

C	Python
<p>Purpose: Total control of the computer, at the expense of more detailed programming.</p> <p>Restrictions: None. C gives you direct access to all computer resources.</p> <p>Versions: "K&R" (1978), C89, C90, C11</p>	<p>Purpose: Fast programming, at the expense of efficiency.</p> <p>Restrictions: You don't have control of memory or direct access to RAM.</p> <p>Versions: 1.0 (Jan 1994), 2.0 (Oct 2000), 3.0 (Dec 2008), 3.6 (Dec 2016)</p>
<p>Comment – text that is removed from a program file during pre-processing.</p> <pre>// Single line comment is all text after a double slash. /* Multi-line comments are any text inside /* and */</pre> <p>Docstring – a multiline comment that describes the purpose, parameters, and return value of a function. In CLion type <code>/**<enter></code> to create a docstring.</p> <pre>/** * Describe the function's purpose. */</pre>	<p>Comment – text ignored by the Python interpreter</p> <pre># Single line comment ''' Multi-line comments are any text inside triple quotes. '''</pre> <p>Docstring – a multiline comment that describes the purpose, parameters, and return value of a function. In PyCharm type <code>"""<enter></code> to create a docstring.</p> <pre>""" Explain what the function does ... :param alpha: alpha is ... :return: the result of ... """</pre>
<p>Variable – a storage location whose value can change.</p> <ul style="list-style-type: none"> Name – Starts with letter, contains only a-z, A-Z, 0-9, _ Data Type – char, int, short int, long int, float, double, ... Value – (depends on its data type) <p><i>Static typing:</i> A variable must be declared and assigned a data type before it can be used.</p> <p>A variable's datatype is determined by its declaration.</p> <p>A variable's datatype never changes.</p> <p>A variable is assigned a memory location at compile time and its location never changes while the program is running.</p>	<p>Variable – a storage location whose value can change.</p> <ul style="list-style-type: none"> Name – Starts with letter, contains only a-z, A-Z, 0-9, _ Data Type – int, float, str, object Value – (depends on its data type) <p><i>Dynamic typing:</i> A variable is automatically created the first time it is used. A variable's datatype is determined by its current value. A variable's datatype can change. A variable is assigned a memory location at run-time and its location can change while the program runs.</p>

<p>Data Type – a description of a memory value</p> <p>C is not an object-oriented language. There are no <i>objects</i> in C.</p> <table><tr><td>character (1 byte)</td><td>char</td></tr><tr><td>integer (4 bytes)</td><td>int</td></tr><tr><td>real number (single, 4 bytes)</td><td>float</td></tr><tr><td>real number (double precision, 8 bytes)</td><td>double</td></tr><tr><td>short (16 bit integer)</td><td>short</td></tr><tr><td>long (32 bit integer)</td><td>long</td></tr><tr><td>double long (64 bit integer)</td><td>long long</td></tr><tr><td>positive or negative</td><td>signed</td></tr><tr><td>non-negative modulo 2m</td><td>unsigned</td></tr><tr><td>pointer to type</td><td>type*</td></tr><tr><td>enumeration constant</td><td>enum tag {name1=value1, ... };</td></tr><tr><td>constant (read-only) value</td><td>type const name;</td></tr><tr><td>no value</td><td>void</td></tr><tr><td>create new name for data type</td><td>typedef type name;</td></tr></table>	character (1 byte)	char	integer (4 bytes)	int	real number (single, 4 bytes)	float	real number (double precision, 8 bytes)	double	short (16 bit integer)	short	long (32 bit integer)	long	double long (64 bit integer)	long long	positive or negative	signed	non-negative modulo 2m	unsigned	pointer to type	type*	enumeration constant	enum tag {name1=value1, ... };	constant (read-only) value	type const name;	no value	void	create new name for data type	typedef type name;	<p>Data Type – a description of a memory value</p> <p>All values in Python are <i>objects</i></p> <p>Simple data types are:</p> <table><tr><td>integer (infinite precision)</td><td>int</td></tr><tr><td>real number (infinite precision)</td><td>float</td></tr></table>	integer (infinite precision)	int	real number (infinite precision)	float
character (1 byte)	char																																
integer (4 bytes)	int																																
real number (single, 4 bytes)	float																																
real number (double precision, 8 bytes)	double																																
short (16 bit integer)	short																																
long (32 bit integer)	long																																
double long (64 bit integer)	long long																																
positive or negative	signed																																
non-negative modulo 2m	unsigned																																
pointer to type	type*																																
enumeration constant	enum tag {name1=value1, ... };																																
constant (read-only) value	type const name;																																
no value	void																																
create new name for data type	typedef type name;																																
integer (infinite precision)	int																																
real number (infinite precision)	float																																
<p>Aggregate Data Types – a collection of values</p> <table><tr><td>array (all elements of the same data type)</td><td>type name[size] ;</td></tr><tr><td>structure (each element can be of a different data type)</td><td>struct name { datatype field1Name; datatype field2Name; ... };</td></tr></table>	array (all elements of the same data type)	type name[size] ;	structure (each element can be of a different data type)	struct name { datatype field1Name; datatype field2Name; ... };	<p>Aggregate Data Types – a collection of values</p> <table><tr><td>string (zero or more characters)</td><td>str</td><td>" "</td></tr><tr><td>list (zero or more elements)</td><td>list</td><td>[]</td></tr><tr><td>tuple (zero or more elements)</td><td>tuple</td><td>()</td></tr><tr><td>dictionary (key à value pairs)</td><td>dict</td><td>{ }</td></tr><tr><td>set (membership)</td><td>set</td><td>()</td></tr><tr><td>file</td><td>file</td><td></td></tr></table>	string (zero or more characters)	str	" "	list (zero or more elements)	list	[]	tuple (zero or more elements)	tuple	()	dictionary (key à value pairs)	dict	{ }	set (membership)	set	()	file	file											
array (all elements of the same data type)	type name[size] ;																																
structure (each element can be of a different data type)	struct name { datatype field1Name; datatype field2Name; ... };																																
string (zero or more characters)	str	" "																															
list (zero or more elements)	list	[]																															
tuple (zero or more elements)	tuple	()																															
dictionary (key à value pairs)	dict	{ }																															
set (membership)	set	()																															
file	file																																
<p>Declaration – create a new memory location with a specific data type</p> <pre>type variableName; // contains a "garbage value"</pre> <p>type variableName = initialValue;</p> <p>type const constantName = initialValue;</p> <p>RULE: All variables, constants, functions, etc. must be declared before they can be used.</p>	<p>Declaration – (Python has no concept of declaring the data type of a variable)</p>																																

<p>Scope – Determines the places in a program where a value can be used. C has 4 scopes:</p> <table border="0"> <tr> <td>Global to program, value was declared in another file</td> <td><code>extern</code></td> </tr> <tr> <td>Global to file, can't be accessed outside the file's code</td> <td><code>static</code></td> </tr> <tr> <td>Local to function</td> <td></td> </tr> <tr> <td>Local to function, persistent between calls</td> <td><code>static</code></td> </tr> </table>	Global to program, value was declared in another file	<code>extern</code>	Global to file, can't be accessed outside the file's code	<code>static</code>	Local to function		Local to function, persistent between calls	<code>static</code>	<p>Scope – Determines the places in a program where a value can be used. Python has two scopes:</p> <ul style="list-style-type: none"> Local to a function. Global to your program. 																																		
Global to program, value was declared in another file	<code>extern</code>																																										
Global to file, can't be accessed outside the file's code	<code>static</code>																																										
Local to function																																											
Local to function, persistent between calls	<code>static</code>																																										
<p>Expression – a set of values and operators that evaluate to a single value.</p> <table border="0"> <tr> <td>struct member operator</td> <td><code>name.member</code></td> </tr> <tr> <td>struct member through pointer</td> <td><code>pointer->member</code></td> </tr> <tr> <td>increment, decrement</td> <td><code>++, --</code></td> </tr> <tr> <td>plus, minus, logical not, bitwise not</td> <td><code>+, -, !, ~</code></td> </tr> <tr> <td>indirection via pointer, address of object</td> <td><code>*pointer, &name</code></td> </tr> <tr> <td>cast expression to type</td> <td><code>(type) expr</code></td> </tr> <tr> <td>size of an object</td> <td><code>sizeof</code></td> </tr> <tr> <td>multiply, divide, modulus (remainder)</td> <td><code>*, /, %</code></td> </tr> <tr> <td>add, subtract</td> <td><code>+, -</code></td> </tr> <tr> <td>left, right shift [bit ops]</td> <td><code><<, >></code></td> </tr> <tr> <td>relational comparisons</td> <td><code>>, >=, <, <=</code></td> </tr> <tr> <td>equality comparisons</td> <td><code>==, !=</code></td> </tr> <tr> <td>and [bit op]</td> <td><code>&</code></td> </tr> <tr> <td>exclusive or [bit op]</td> <td><code>^</code></td> </tr> <tr> <td>or (inclusive) [bit op]</td> <td><code> </code></td> </tr> <tr> <td>logical and</td> <td><code>&&</code></td> </tr> <tr> <td>logical or</td> <td><code> </code></td> </tr> <tr> <td>conditional expression</td> <td><code>expr1 ? expr2 :</code></td> </tr> <tr> <td><code>expr3</code></td> <td></td> </tr> <tr> <td>assignment operators</td> <td><code>+=, -=, *=, ...</code></td> </tr> <tr> <td>expression evaluation separator</td> <td><code>,</code></td> </tr> </table> <p>Unary operators, conditional expression and assignment operators group right to left; all others group left to right.</p>	struct member operator	<code>name.member</code>	struct member through pointer	<code>pointer->member</code>	increment, decrement	<code>++, --</code>	plus, minus, logical not, bitwise not	<code>+, -, !, ~</code>	indirection via pointer, address of object	<code>*pointer, &name</code>	cast expression to type	<code>(type) expr</code>	size of an object	<code>sizeof</code>	multiply, divide, modulus (remainder)	<code>*, /, %</code>	add, subtract	<code>+, -</code>	left, right shift [bit ops]	<code><<, >></code>	relational comparisons	<code>>, >=, <, <=</code>	equality comparisons	<code>==, !=</code>	and [bit op]	<code>&</code>	exclusive or [bit op]	<code>^</code>	or (inclusive) [bit op]	<code> </code>	logical and	<code>&&</code>	logical or	<code> </code>	conditional expression	<code>expr1 ? expr2 :</code>	<code>expr3</code>		assignment operators	<code>+=, -=, *=, ...</code>	expression evaluation separator	<code>,</code>	<p>Expression – a set of values and operators that evaluate to a single value. <i>operand operator operand</i>, e.g., $(37 + \text{alpha}) / (2 ** 3)$</p> <ul style="list-style-type: none"> operators: <code>+</code>, <code>-</code>, <code>*</code>, <code>/</code>, <code>//</code>, <code>**</code> order of operations: <code>()</code>, <code>**</code>, <code>*</code>, <code>/</code>, <code>//</code>, <code>+</code>, <code>-</code> operators are context sensitive: <code>"ab" + "def"</code> different from <code>(37 + 12)</code> <p>Note: C does <u>not</u> have an exponentiation operator, use the <code>pow()</code> function.</p>
struct member operator	<code>name.member</code>																																										
struct member through pointer	<code>pointer->member</code>																																										
increment, decrement	<code>++, --</code>																																										
plus, minus, logical not, bitwise not	<code>+, -, !, ~</code>																																										
indirection via pointer, address of object	<code>*pointer, &name</code>																																										
cast expression to type	<code>(type) expr</code>																																										
size of an object	<code>sizeof</code>																																										
multiply, divide, modulus (remainder)	<code>*, /, %</code>																																										
add, subtract	<code>+, -</code>																																										
left, right shift [bit ops]	<code><<, >></code>																																										
relational comparisons	<code>>, >=, <, <=</code>																																										
equality comparisons	<code>==, !=</code>																																										
and [bit op]	<code>&</code>																																										
exclusive or [bit op]	<code>^</code>																																										
or (inclusive) [bit op]	<code> </code>																																										
logical and	<code>&&</code>																																										
logical or	<code> </code>																																										
conditional expression	<code>expr1 ? expr2 :</code>																																										
<code>expr3</code>																																											
assignment operators	<code>+=, -=, *=, ...</code>																																										
expression evaluation separator	<code>,</code>																																										
<p>Statement – a command given to the computer. C is totally free form, meaning newlines have no meaning. A single line in a C code file can have 0 to an infinite number of statements. Statements must be separated by a semicolon (;). Statements inside braces, { ... }, form a statement block.</p>	<p>Statement – a command given to the computer. Python code is organized one statement per line.</p> <p>Statements at the same level of indentation form a statement block</p>																																										

<p>Assignment Statement – set the value of a variable. <i>variable = expression</i></p>	<p>Assignment Statement – set the value of a variable. <i>variable = expression</i></p>
<p>Selection - conditional execution of a group of statements.</p> <pre> if (expression) { statement(s) # if expression is true } if (expression) { statement(s) # if expression is true } else { statement(s) # if expression is false } if (expression1) { statement(s) # if expression1 is true } else if (expression2) { statement(s) # if expression1 is false and expression2 is true } else if (expression3) { statement(s) # if expression1 and expression 2 are false and expression3 is true } else { statement(s) # if expression1 and expression2 and expression3 are all false } switch (expression) { case const1: statement(s); break; case const2: statement(s); break; default: statement(s) } </pre>	<p>Selection - conditional execution of a group of statements.</p> <pre> if expression: statement(s) # if expression is true if expression: statement(s) # if expression is true else: statement(s) # if expression is false if expression1: statement(s) # if expression1 is true elif expression2: statement(s) # if expression2 is true elif expression3: statement(s) # if expression3 is true else: statement(s) Python does not have a statement equivalent to a switch statement </pre>
<p>Iteration – repeatedly execute a group of statements.</p> <p>for loop – general format</p> <pre> for (initialize; test_for_continue; modification) { ... } </pre> <p>(Note: There is no such thing as an <i>iterable</i> object in C.)</p>	<p>Iteration – repeatedly execute a group of statements.</p> <p>for loop – general format</p> <pre> for variable in iterable_object: statement(s) </pre>

<pre> for (int j=0; j<10; j++) { ... } for (int j=20; j<30; j++) { ... } for (int j=-5; j<10; j+=2) { ... } while loop while (expression) { ... } repeat-until loop do { ... } while(expression); </pre>	<pre> for <i>variable</i> in range(10): ... for <i>variable</i> in range(20,30): ... for <i>variable</i> in range(-5,10,2): ... while loop while expression: ... repeat-until loop while True: ... if <i>expression</i>: break </pre>
<p>Function Definition – create a group of statements to be executed as a unit.</p> <pre> datatype <i>name</i>(<i>parameters</i>) { <i>statement(s)</i> } </pre>	<p>Function Definition – create a group of statements to be executed as a unit.</p> <pre> def <i>name</i>(<i>parameters</i>): <i>statement(s)</i> ... </pre>
<p>Import Statement – load the variables and functions defined in another file.</p> <pre> include library file #include <name> include user file #include "name" </pre>	<p>Import Statement – load the variables and functions defined in another file.</p> <pre> import <i>filename</i> - import all definitions in the specified file from <i>filename</i> import <i>name</i> - import the specified items from <i>filename</i> import <i>name</i> as <i>newName</i> - import and rename </pre>
<p>Console input – allow a user to input a value from the console</p> <pre> scanf(formatString, addressOfVariable) printf("Enter an integer number: "); scanf("%d", &value); </pre>	<p>Console input – allow a user to input a value from the console</p> <pre> <i>variable</i> = input(" <i>prompt</i> ") </pre>

Codes for Formatted output - format specifier: %-+ 0w:pmc

-	left justify		
+	print with sign		
space	print space if no sign		
0	pad with leading zeros		
w	minimum field width		
p	precision		
m	conversion character: h short, l long, L long double		
c	conversion character:		
	d,i integer	u	unsigned
	o octal	x,X	hexadecimal
	c single char	s	char string
	f float	e,E	exponential
	p pointer	lf	double

Example:

```
printf("Fred weights %.1f lbs and is %d years old",  
      weight, age)
```

Formatted output: format specifications:

string output:

{:10} – print exactly 10 characters, (default: left justified)

{:<10} – print exactly 10 characters, left justified.

{:>12} – print exactly 12 characters, right justified.

{:^8} – print exactly 8 characters, center justified.

int output:

{:10d} – print decimal integer using exactly 10 characters, right justified.

{:8b} – print binary integer using exactly 8 characters, right justified.

{:6x} – print hexadecimal integer using exactly 6 characters, right justified.

float output:

{:10.2f} – exactly 2 digits after the decimal point; use exactly 10 characters,

{:12g} – significant digits after the decimal point; use exactly 12 characters,

{:8.2e} – exponential notation; 2 digits accuracy; use exactly 8 characters,

Example:

```
print("Fred weights {:.1f} lbs and is {:d} years old"  
      .format(weight, age))
```

Example Program

```
// =====  
// A simple example C program that finds prime numbers.  
// Created by Dr. Wayne Brown on 1/3/2017.  
// =====  
  
#include <math.h>  
#include <stdio.h>  
  
#define TRUE    1  
#define FALSE   0  
  
// -----  
int is_prime(int n) {  
    /**  
     * Determines if the value of n is prime.  
     * Returns True if n is prime, False otherwise  
     */  
    int max_divisor = (int) ceil(sqrt(n));  
    for (int divisor = 2; divisor <= max_divisor; divisor++) {  
        if (n % divisor == 0) {  
            return FALSE;  
        }  
    }  
  
    return TRUE;  
}  
  
// -----  
int main() {  
    // Print all of the prime numbers <= 100  
    for (int value = 3; value <= 100; value++) {  
        if (is_prime(value)) {  
            printf("%d\n", value);  
        }  
    }  
}
```

Example Program

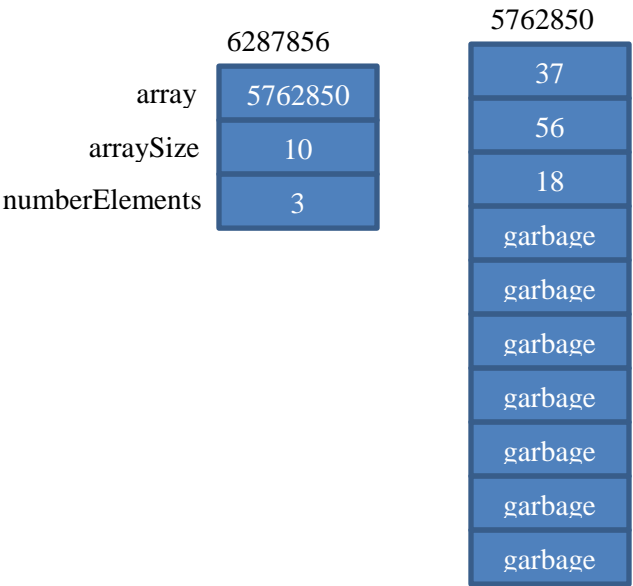
```
#!/usr/bin/env python  
  
"""  
A simple example Python program that finds prime numbers.  
"""  
#  
=====  
from math import sqrt, ceil  
  
# Metadata  
__author__ = "Wayne Brown"  
  
# -----  
def is_prime(n):  
    """  
    Determines if the value of n is prime  
    :param n: is an integer  
    :return: True if n is prime, false otherwise  
    """  
    for divisor in range(2, ceil(sqrt(n))):  
        if n % divisor == 0:  
            return False  
  
    return True  
  
# -----  
def main():  
    # Print all of the prime numbers <= 100  
    for value in range(3, 101):  
        if is_prime(value):  
            print(value)  
  
# -----  
if __name__ == "__main__":  
    main()
```

The remainder of these pages describes only C syntax.

A variable has a *memory address* and a *value*. In the following diagrams, a variable is represented as a box. Inside the box is the variable's *value*. The variable's *address* is displayed above the box. The variable's *name* is displayed to the left of the box. For example:

	Address
Name	Value

A visual diagram of the memory for an ArrayList:



A C description of an ArrayList:

```
// Define the data type for each element of the list.
typedef int ElementType;

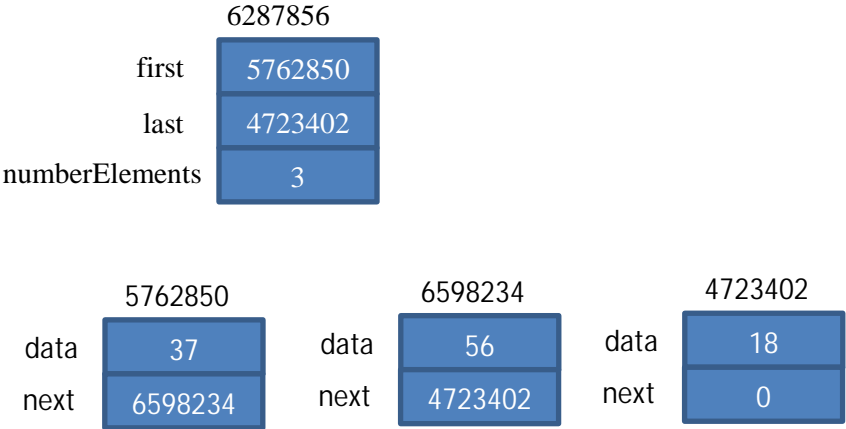
// Define the structure that holds the meta-data for one list.
typedef struct ArrayList {
    ElementType *array;
    int arraySize;
    int numberElements;
} ArrayList;

// Create the structure that holds the list information
ArrayList * myList = (ArrayList *) malloc(sizeof(ArrayList));

// Allocate a block of memory to hold the elements
myList->array = (ElementType *) malloc(sizeof(ElementType) * 10);
```

C Syntax:	Evaluates to these values for the example to the left:
myList	6287856
myList->array	5762850
myList->arraySize	10
myList->numberElements	3
myList->array[0]	37
myList->array[1]	56
myList->array[2]	18
myList->array[3]	(garbage)

A visual diagram of the memory for a LinkedList:



A C description of a LinkedList:

```
// Define the data type for each element of the list.
typedef int ElementType;

// Define on node of the linked list
typedef struct node {
    ElementType data;
    struct node * next;
} Node;

// Define the meta-data that stores the linked list.
typedef struct linkedList {
    Node * first;
    Node * last;
    int numberElements;
} LinkedList;

// Create the structure that holds the list information
LinkedList * myList = (LinkedList *) malloc(sizeof(LinkedList));

// Allocate a block of memory to hold one node of the list
Node * oneNode = (Node *) malloc(sizeof(Node));
```

C Syntax:	Evaluates to these values for the example to the left:
myList	6287856
myList->first	5762850
myList->last	4723402
myList->numberElements	3
myList->first->data	37
myList->first->next	6598234
myList->last->data	18
myList->last->next	0
Node * aNode;	(aNode contains garbage)
aNode = myList->first	5762850
aNode->data	37
aNode->next	6598234
aNode->next->data	56

Using the CLion debugger:

- To see the contents of an array if you only have a pointer to the array, in the "variables panel" create a new "watch" and cast the pointer to an array.
 - For example, if you have declared: `TypeXYZ *alpha;` and `alpha` points to a valid memory block (because you malloc'ed memory or assigned it to an existing array), then the CLion *watch* would be `(TypeXYZ (*) [size]) alpha`, where `size` is the number of elements you want CLion to be able to display.

bit manipulation in C:		bit manipulation in Python: (Identical to C syntax)	
Operator	What it does:	Operator	What it does:
<code>a << n</code>	bit-wise left shift the bits in <code>a</code> by <code>n</code> bits	<code>a << n</code>	bit-wise left shift the bits in <code>a</code> by <code>n</code> bits
<code>a >> n</code>	bit-wise right shift the bits in <code>a</code> by <code>n</code> bits	<code>a >> n</code>	bit-wise right shift the bits in <code>a</code> by <code>n</code> bits
<code>a & b</code>	bit-wise logical AND on corresponding bits in <code>a</code> and <code>b</code>	<code>a & b</code>	bit-wise logical AND on corresponding bits in <code>a</code> and <code>b</code>
<code>a b</code>	bit-wise logical OR on corresponding bits in <code>a</code> and <code>b</code>	<code>a b</code>	bit-wise logical OR on corresponding bits in <code>a</code> and <code>b</code>
<code>a ^ b</code>	bit-wise logical XOR on corresponding bits in <code>a</code> and <code>b</code>	<code>a ^ b</code>	bit-wise logical XOR on corresponding bits in <code>a</code> and <code>b</code>
<code>~a</code>	bit-wise logical NOT (complement) bits in <code>a</code>	<code>~a</code>	bit-wise logical NOT (complement) bits in <code>a</code>

Static arrays

- Arrays created by the compiler.
- The size of the array must be known at compile time. (Arrays inside functions can have a size that is determined by a function parameter.)
- The memory for local variables is allocated on the run-time stack when a function is called, and automatically reclaimed when a function exits.

Create	Access using subscript notation	Access using pointer notation	Notes:
<pre>int a[100]; // 10 rows, 20 columns int b[10][20]; // 5 slices, 6 rows, 10 columns int c[5][6][10];</pre>	<pre>a[3] = 5; a[2] = a[4] + a[6]; // change row 2, column 3 b[2][3] = 5; // change slice 2, row 1, // column 5 c[2][1][5] = 5;</pre>	<pre>*(a+3) = 5; *(a+2) = *(a+4) + *(a+6) // change row 2, column 3 *(b + 2*20 + 3) = 5; // change slice 2, row 1, // column 5 *(c + 2*60 + 1*10 + 5) = 5;</pre>	<ul style="list-style-type: none"> • The array name translates into the address of the array's memory. • The array name is not a variable; it can't be changed to point to a different block of memory. • Arrays are always stored in row-major order.

Dynamic arrays

- Arrays created at run-time by the program.
- The array size does not need to be known at compile time.
- The memory block exists until it is returned to the "memory pool" using `free(address)`.

Create	Access using subscript notation	Access using pointer notation	Notes:
<pre>// 1D array int *a = malloc(100 * sizeof(int)); // 2D array as a single block of // memory; 10 rows, 20 columns int *b = malloc(10 * 20 * sizeof(int)); // 2D array as an array of pointers, // each to a single row. 10 rows // and 20 columns int **c = malloc(sizeof(int *) * 10); for (int j=0; j<10; j++) c[j] = malloc(sizeof(int) * 20);</pre>	<pre>a[3] = 5; a[2] = a[4] + a[6]; // set row 2, column 3 b[2*20 + 3] = 5; c[2][5] = 5;</pre>	<pre>*(a+3) = 5; *(a+2) = *(a+4) + *(a+6) // set row 2, column 3 *(b + 2*20 + 3) = 5; *(*(c + 2) + 5) = 5;</pre>	<ul style="list-style-type: none"> • The pointer can be treated as an array. • The pointer is a variable; it can be changed to point to a different block of memory. • Arrays are always stored in row-major order.