#Pointer array

#include<iostream>

using namespace std;

int main () {

// Define an integer array of size 10 and initialize some elements

int arr[10] = {23, 122, 41, 67};

// Printing the address of the first memory block of the array

cout <<" address of first memory block is " << arr << endl;

cout << arr[0] << endl; // Printing the first element

cout <<" address of first memory block is " << &arr[0] << endl;

// Demonstrating pointer arithmetic and dereferencing

cout << "4th " << \*arr << endl; // Value at the first element (same as arr[0])

cout << "5th " << \*arr + 1 << endl; // First element value plus 1 (not next element)

cout << "6th " << \*(arr + 1) << endl; // Value at the second element (arr[1])

cout << "7th " << \*(arr) + 1 << endl; // Value at the first element plus 1

cout << "8th " << arr[2] << endl; // Value at the third element

cout << "9th " << \*(arr+2) << endl; // Another way to get the value at the third element

int i = 3;

cout << i[arr] << endl; // Unusual syntax, same as arr[i]

// Exploring the size of the array and its pointer

int temp[10] = {1,2};

cout << sizeof(temp) << endl; // Size of the array (40 bytes, 10 \* 4 bytes for int)

cout << " 1st " << sizeof(\*temp) << endl; // Size of the first element (4 bytes)

cout << " 2nd " << sizeof(&temp) << endl; // Size of the pointer to the array (likely 8 bytes on a 64-bit system)

int \*ptr = &temp[0];

cout << sizeof(ptr) << endl ; // Size of the pointer (8 bytes)

cout << sizeof(\*ptr) << endl ; // Size of the value pointed to (4 bytes)

cout << sizeof(&ptr) << endl ; // Size of the pointer to the pointer (8 bytes)

// Exploring pointer behavior with arrays

int a[20] = {1,2,3,5};

cout << "-> " << &a[0] << endl; // Address of the first element of array `a`

cout << &a << endl; // Address of the entire array (same as &a[0])

cout << a << endl; // Address of the first element (same as &a[0])

int \*p = &a[0];

cout << p << endl; // Address of the first element of `a`

cout << \*p << endl; // Value of the first element of `a`

cout << "-> " << &p << endl; // Address of the pointer `p`

// ERROR: Cannot perform this operation because `arr` is a constant pointer

//arr = arr+1;

int \*ptr = &arr[0];

cout << ptr << endl; // Address of the first element of `arr`

ptr = ptr + 1;

cout << ptr << endl; // Address of the second element of `arr`

return 0;

}

**Detailed Breakdown:**

1. **Array and Memory Addressing**:
   * arr and &arr[0] both give the address of the first element of the array. This is because in the context of an array, arr is a constant pointer to the first element.
   * \*arr gives the value of the first element. When you add to arr (like \*(arr + 1)), you move to subsequent elements of the array.
2. **Pointer Arithmetic**:
   * Pointer arithmetic allows you to move through the elements of an array using the pointer. For example, \*(arr + 1) gives you the value of the second element, arr[1].
3. **Unusual Array Indexing**:
   * i[arr] is the same as arr[i]. This is because array access is defined as \*(arr + i), and due to commutativity of addition, \*(arr + i) is the same as \*(i + arr).
4. **Sizeof Operations**:
   * sizeof(temp) gives the total size of the array in bytes (e.g., 40 bytes for an array of 10 integers).
   * sizeof(\*temp) gives the size of a single integer (4 bytes).
   * sizeof(ptr) gives the size of the pointer itself (8 bytes on a 64-bit system).
   * sizeof(&ptr) gives the size of the pointer to the pointer (again, 8 bytes).
5. **Pointer Copying and Address Manipulation**:
   * You can copy a pointer and both will point to the same memory address. The original data they point to can be modified through either pointer.
   * ptr = ptr + 1; moves the pointer to the next integer in the array.
6. **Error with Array Pointer Modification**:
   * The line arr = arr + 1; is commented out because it would generate an error. arr is a constant pointer (it always points to the start of the array), so you cannot reassign it.

**Key Concepts:**

* **Pointer and Array Relationship**:
  + An array name (like arr) acts as a pointer to its first element, but it is a constant pointer and cannot be changed.
* **Pointer Arithmetic**:
  + You can use pointer arithmetic to navigate through an array.
* **Memory Addresses**:
  + Pointers store memory addresses. The & operator retrieves the address of a variable, and \* (dereference operator) retrieves the value at that address.
* **Pointer Size**:
  + On a 64-bit system, pointers generally have a size of 8 bytes, regardless of the data type they point to.

#pointer\_characterArray.

#include<iostream>

using namespace std;

int main() {

int arr[5] = {1,2,3,4,5}; // An array of integers

char ch[6] = "abcde"; // A character array (string) with a null terminator

// Printing the integer array's name, which decays to a pointer to the first element

cout << arr << endl;

// Printing the character array's name

cout << ch << endl;

char \*c = &ch[0]; // Pointer to the first character of the array

cout << c << endl; // Since `c` is a pointer to char, it prints the entire string

char temp = 'z'; // A character variable

char \*p = &temp; // Pointer to the character variable

// This prints the character 'z' followed by whatever garbage characters might be in memory

cout << p << endl;

return 0;

}

**Detailed Breakdown:**

1. **Printing arr (Integer Array)**:
   * cout << arr << endl;
   * Here, arr is the name of the integer array, and when you try to print it, arr decays to a pointer to its first element. Since arr is an integer pointer (int\*), printing it directly with cout will display the memory address of the first element in the array, not the contents of the array itself.
2. **Printing ch (Character Array)**:
   * cout << ch << endl;
   * ch is a character array initialized as "abcde". In C++, when you print a character array with cout, it doesn't print the memory address but instead prints the entire string stored in the array. This happens because ch is treated as a pointer to a null-terminated string (char\*), and cout is overloaded to print strings when a char\* is passed.
3. **Printing c (Pointer to Character Array)**:
   * char \*c = &ch[0];
   * cout << c << endl;
   * c is a pointer to the first character of the ch array. When you print c, it behaves just like printing ch because c points to the same memory location. Thus, it prints the entire string "abcde".
4. **Printing p (Pointer to Single Character)**:
   * char temp = 'z';
   * char \*p = &temp;
   * cout << p << endl;
   * p is a pointer to the single character temp. When you try to print p, it will start at the address of temp and attempt to print a null-terminated string starting from there. Since temp is just a single character and not part of a string, cout will print the character z followed by whatever characters happen to be in memory until it finds a null terminator (\0). This can lead to printing garbage characters.

**Key Concepts:**

* **Array Name as a Pointer**:
  + The name of an array (arr or ch) is essentially a constant pointer to the first element of the array. However, the way this pointer is interpreted by cout depends on the data type.
* **Printing Pointers**:
  + When you print an int\*, cout shows the memory address.
  + When you print a char\*, cout interprets it as a C-string and prints the sequence of characters until a null terminator is encountered.
* **Character Pointers and Strings**:
  + Printing a char\* that points to a single character (not part of a null-terminated string) can lead to unexpected results, including printing garbage values.

#pointer\_functions

#include<iostream>

using namespace std;

void print(int \*p) {

cout << \*p << endl;

}

void update(int \*p) {

// p = p + 1;

// cout << "inside "<< p <<endl;

\*p = \*p + 1;

}

int getSum(int \*arr, int n) {

cout << endl << "Size : " << sizeof(arr) << endl;

int sum = 0;

for(int i=0; i<n; i++) {

sum += arr[i];

}

return sum;

}

int main() {

/\*

int value = 5;

int \*p = &value;

// print(p);

cout <<" Before " << \*p << endl;

update(p);

cout <<" After " << \*p << endl;

\*/

int arr[6] = {1,2,3,4,5,8};

cout << "Sum is " << getSum(arr+3, 3) << endl ;

return 0;

}

**Detailed Explanation:**

1. **The print Function**:
   * **Purpose**: Prints the value pointed to by the pointer p.
   * **How It Works**: The function accepts a pointer to an integer, dereferences it using \*p, and then prints the value.
2. **The update Function**:
   * **Purpose**: Modifies the value of the integer pointed to by the pointer p.
   * **How It Works**:
     + The line p = p + 1; (commented out) would change the address stored in p to the next integer's address in memory, but this would only affect the local copy of p inside this function, not the original pointer passed from the caller.
     + The line \*p = \*p + 1; increments the value stored at the memory location pointed to by p.
3. **The getSum Function**:
   * **Purpose**: Calculates the sum of n elements of an integer array starting from the location pointed to by arr.
   * **How It Works**:
     + The function accepts a pointer to the first element in the array (int \*arr) and the number of elements (int n) to sum.
     + The sizeof(arr) prints the size of the pointer arr, which is typically 8 bytes on a 64-bit system, regardless of the array size. This illustrates that in C++, when you pass an array to a function, it decays to a pointer, losing the original array's size information.
     + The for loop iterates n times, adding each array element to sum, which is then returned.
4. **The main Function**:
   * **Purpose**: Demonstrates the usage of the above functions.
   * **How It Works**:
     + The commented-out section shows how to use the print and update functions. It prints the value before and after updating it via a pointer.
     + The getSum function is called with arr+3 as the pointer, which means the function starts summing from the 4th element (index 3) of the array arr. The second argument 3 tells the function to sum three elements from that point onwards.
     + Since arr+3 points to the element 4, the function sums 4, 5, and 8, resulting in 17.

**Output:**

Given the code, the output will be:

csharp

Copy code

Size : 8

Sum is 17

* **Size : 8**: This indicates that the size of the arr pointer (which is passed to getSum) is 8 bytes, which is typical for a pointer on a 64-bit system.
* **Sum is 17**: The sum of elements 4, 5, and 8 is correctly calculated and displayed.

This code illustrates essential concepts of pointers in C++, such as pointer arithmetic, passing pointers to functions, and how arrays behave when passed to functions.