Computer Systems, et al.

Vassilis Markos, Mediterranean College

Week 04

Contents

Pointers

Can You Predict The Output?

```cpp
// source/stars_005.cpp
#include <iostream>
using namespace std;

int main() {
    int x;
    cout << "Enter an int: ";
    cin >> x;
    int *ptrx = &x;
    (*ptrx)++;
    cout << x << endl;
}
```

Variations On A Theme: What Will This Print?

```cpp
// source/stars_005a.cpp
#include <iostream>
using namespace std;

int main() {
    int x;
    cout << "Enter an int: ";
    cin >> x;
    int *ptrx = &x;
    int y = (*ptrx++);
    auto z = (*++ptrx);
    cout << x << "\n" << y << "\n" << z << endl;
}
```

A Typical Output

For user input `7` the above program might print:

```
Enter an int: 7
7
7
22002
```

Can you explain why?

Pointer Arithmetic

- Since the user has entered `7`, the value of `x` is `7`, so this is the first of the two sevens.
- The second seven is the value stored at `y` through the evaluation of the expression `(*ptrx++)`:
  - `*ptrx++` first uses the current value of `ptrx`, which is where `x` is stored in RAM and then increments `ptrx` by "1".
  - Since `ptrx` is a pointer to an `int`, "increment by 1" in this case is interpreted as "increment by **4 bytes**", i.e., the size of an `int`.
  - So, in this case, `y` contains the value of `x`, which is what `ptrx` points to.

Pointer Arithmetic

- The value of `z` is determined through the expression `*++ptrx`.
- This means that:
  - `++ptrx` **first increments** the pointer by 4 bytes and then;
  - **dereferences** the incremented value.
  - So, C++ is now trying to interpret what is stored in the corresponding memory locations, which results, in our case, to `22002`.
  - Note that what is kept in those locations is not something we can predict, as now `ptrx` points to a memory location that we have not assigned ourselves a value. So, each time you execute this program, something different should be printed on your console.

A Brief Note About `auto`

In the above, we declared `z` as `auto`:

- `auto` is originally a C thing but there it was virtually useless, as discussed in the C Infrequently Asked Questions (IAQ: `https://www.seebs.net/faqs/c-iaq.html`).
- Since C++ 11 it is actually useful when it comes to declaring a complex variable type which can, however, be inferred by the compiler at compilation time.
- One common usage example is with C++ Templates, which, however, will not bother us (Phew!).

Playing Around a Bit

We can be more specific and avoid using `auto`, as shown below:

```
1  // source/stars_005b.cpp
2  #include <iostream>
3  using namespace std;
4
5  int main() {
6      int x;
7      cout << "Enter an int: ";
8      cin >> x;
9      int *ptrx = &x;
10     cout << "ptrx: " << ptrx << endl;
11     ++ptrx;
12     double y = *ptrx;
13     cout << x << "\n" << y << endl;
14     cout << "ptrx: " << ptrx << endl;
15 }
```

Playing Around a Bit

A typical output should be something like this:

```
Enter an int: 7
ptrx: 0x7ffe50cc8188
7
21967
ptrx: 0x7ffe50cc818c
```

Here, `y` is declared as an `int`, so C++ prints an `int` as read from the 4 bytes stored at memory locations: `0x7ffe50cc818c`, `0x7ffe50cc818d`, `0x7ffe50cc818e`, `0x7ffe50cc818f`.

More Quirks

- In `source/stars_005b.cpp` change in line 12 the type declaration of `y` from `int` to `double`.
- What do you observe when you run the program?
- Now, a typical value of `y` looks somewhat like: `-5.48475e+07`.
- Can you explain this behaviour?

More Quirks

- In `source/stars_005b.cpp` change in line 12 the type declaration of `y` from `int` to `double`.
- What do you observe when you run the program?
- Now, a typical value of `y` looks somewhat like: `-5.48475e+07`.
- Can you explain this behaviour?
- In short, this is because declaring different data types forces C++ to interpret the content at the referenced memory locations in different ways (in principle).

Computer Numbers

- Assume that a computer's memory looks as shown right (memory locations indicated at the bottom of each cell).
- Then, which bytes should the compiler use to read an `int` starting at memory location `0x7ffefcbb17f4`?

01110110 0x7ffefcbb17f4	11111111 0x7ffefcbb17f5
01111001 0x7ffefcbb17f6	10101010 0x7ffefcbb17f7
01111001 0x7ffefcbb17f8	00011110 0x7ffefcbb17f9
10101001 0x7ffefcbb17fa	00000001 0x7ffefcbb17fb

Computer Numbers

- Since an `int` takes up 4 bytes in the computer's memory, the compiler will make use of the first four bytes, as shown right. This results to the integer value: `1996454314`.
- What if we want to read a double?

01110110 0x7ffefcbb17f4	11111111 0x7ffefcbb17f5
01111001 0x7ffefcbb17f6	10101010 0x7ffefcbb17f7
01111001 0x7ffefcbb17f8	00011110 0x7ffefcbb17f9
10101001 0x7ffefcbb17fa	00000001 0x7ffefcbb17fb

Computer Numbers

- Since a `double` takes up 8 bytes in the computer's memory, the compiler will make use of all the bytes, as shown right. This results to the float value: `1.5857893356345228e+265`.
- For more on how bitstrings are converted to numbers:
  - `Two's Complement`.
  - IEEE 754.

01110110 0x7ffefcbb17f4	11111111 0x7ffefcbb17f5
01111001 0x7ffefcbb17f6	10101010 0x7ffefcbb17f7
01111001 0x7ffefcbb17f8	00011110 0x7ffefcbb17f9
10101001 0x7ffefcbb17fa	00000001 0x7ffefcbb17fb

Can You Predict The Output?

```cpp
// source/stars_006.cpp
#include <iostream>
using namespace std;

int main() {
    int x, y;
    cout << "Enter two ints: ";
    cin >> x;
    cin >> y;
    int *ptrx = &x;
    int *ptry = &y;
    ptry = ptrx;
    (*ptrx)--;
    ptry = &y;
    cout << x << ", " << y << endl;
}
```

Can You Predict The Output?

```cpp
// source/stars_007.cpp
#include <iostream>
using namespace std;

int main() {
    int x, *y;
    cout << "Enter an int: ";
    cin >> x;
    y = &x;
    (*y)++;
    cout << x << endl;
}
```

Arrays

Arrays In C++

Can you guess what the following will print?

```cpp
// source/arrays_001.cpp
#include <iostream>
using namespace std;

int main() {
    int arr[3];
    arr[0] = 4;
    arr[1] = 6;
    arr[2] = -5;
    for (int i = 0; i < 3; i++) {
        cout << "arr[" << i << "] == " << arr[i] << endl;
    }
}
```

Array Initialisation

We can also provide array elements all at once, as follows:

```
1 // source/arrays_002.cpp
2 #include <iostream>
3 using namespace std;
4
5 int main() {
6     int arr1[5] = { 4, -2, 0, 4, 6 };
7     int arr2[] = { 6, 5, 7, 9 };
8     cout << "arr1[3] == " << arr1[3] << "\narr2[1] == " <<
      arr2[1] << endl;
9 }
```

Dynamic Initialisation

We can also initialise the values of an array based on others' input (e.g., users, another process):

```cpp
// source/arrays_003.cpp
#include <iostream>
using namespace std;

int main() {
    char arr[3];
    for (int i = 0; i < 3; i++) {
        cout << "Please, enter a character: ";
        cin >> arr[i];
    }
    cout << arr[0] << arr[1] << arr[2] << endl;
}
```

What Will This Print?

```cpp
// source/arrays_004.cpp
#include <iostream>
using namespace std;

int main() {
    char arr[];
    for (int i = 0; i < 3; i++) {
        cout << "Please, enter a character: ";
        cin >> arr[i];
    }
    cout << arr[0] << arr[1] << arr[2] << endl;
}
```

Dynamic Initialisation And Array Size

The above must have printed something along the following lines:

```
arrays_004.cpp: In function ‘int main()’:
arrays_004.cpp:6:10: error: storage size of ‘arr’
isn’t known
6 |     char arr[];
  |          ^~~
```

This actually means that in order to **refer to an array's element by its index** you must **first determine the array's size!**

Pointers And Arrays

What will the following print?

```cpp
// source/arrays_005.cpp
#include <iostream>
using namespace std;

int main() {
    int arr[] = {2, 6, 5, 1};
    int* ptr = arr;
    ptr++;
    cout << *ptr << endl;
}
```

Pointers And Arrays

Do you observe something strange in the following?

```
1 int main() {
2     int arr[] = {2, 6, 5, 1};
3     int* ptr = arr;
4     ptr++;
5     cout << *ptr << endl;
6 }
```

Pointers And Arrays

Do you observe something strange in the following?

```
1 int main() {
2     int arr[] = {2, 6, 5, 1};
3     int* ptr = arr;
4     ptr++;
5     cout << *ptr << endl;
6 }
```

- **Line 3:** We declare an integer pointer and store the **array** there, **not a reference!**

Pointers And Arrays

Do you observe something strange in the following?

```
1 int main() {
2     int arr[] = {2, 6, 5, 1};
3     int* ptr = arr;
4     ptr++;
5     cout << *ptr << endl;
6 }
```

- **Line 3:** We declare an integer pointer and store the **array** there, **not a reference!**
- Why does it work?

Pointers And Arrays

- In C++, arrays of type `<T>` are actually pointers to items of type `<T>`.
- This means that, when declaring an array, we are actually declaring a pointer to the first memory location occupied by its first element.
- So, arrays are actually of **pointer type!**
- This means that using `&` to get their memory address is of no use, since they already represent a memory address.

Pointer Tricks

What will the following print?

```cpp
// source/arrays_006.cpp
#include <iostream>
using namespace std;

int main() {
    int arr[] = {2, 6, 5, 1};
    float* ptr = (float*) arr;
    ptr++;
    cout << *ptr << endl;
}
```

Pointer Tricks

What will the following print?

```
1  // source/arrays_007.cpp
2  #include <iostream>
3  using namespace std;
4
5  int main() {
6      int arr[] = {2, 6, 0, 0, 4};
7      double* ptr = (double*) arr;
8      ptr++;
9      cout << *ptr << endl;
10 }
```

Pointer Tricks

What will the following print?

```
1  // source/arrays_008.cpp
2  #include <iostream>
3  using namespace std;
4
5  int main() {
6      int arr[] = {2, 6, 0, 1, 4};
7      double* ptr = (double*) arr;
8      ptr++;
9      cout << *ptr << endl;
10 }
```

Looping Over An Array

```cpp
// source/arrays_009.cpp
#include <iostream>
using namespace std;

int main() {
    int arr[] = {2, 6, 0, 1, 4};
    for (int i = 0; i < 5; i++) {
        cout << "arr[" << i << "] == " << arr[i] << endl;
    }
}
```

Looping Over An Array With Pointers

Can you loop over the same array without using the `arr[i]` syntax?

Looping Over An Array With Pointers

Can you loop over the same array without using the `arr[i]` syntax?

```cpp
// source/arrays_010.cpp
#include <iostream>
using namespace std;

int main() {
    int arr[] = {2, 6, 0, 1, 4};
    for (int i = 0; i < 5; i++) {
        cout << "arr[" << i << "] == " << *(arr + i) << endl;
    }
}
```

Pointer Arithmetic (Again)

- In general, the expression `pointer + integer` is interpreted as: increment the `pointer` by the size of its pointing type times the `integer`.
- So, for an `int* ptr`, `ptr + 6` should be interpreted as “move the pointer `ptr` by `6 * sizeof(int)`, i.e., `6 * 4` bytes”.
- So, for a `double* ptr`, `ptr + 5` should be interpreted as “move the pointer `ptr` by `5 * sizeof(double)`, i.e., `5 * 8` bytes”.

Passing Arrays To Functions

What will this print?

```cpp
// source/arrays_011.cpp
#include <iostream>
using namespace std;

void printArray(int arr[], int length) {
    for (int i = 0; i < length; i++) {
        cout << "arr[" << i << "] == " << *(arr + i) << endl;
    }
}

int main() {
    int arr[] = {2, 6, 0, 1, 4};
    printArray(arr, 5);
}
```

Passing Arrays To Functions

What will this print?

```cpp
// source/arrays_012.cpp
#include <iostream>
using namespace std;

void printArray(int* arr, int length) {
    for (int i = 0; i < length; i++) {
        cout << "arr[" << i << "] == " << *(arr + i) << endl;
    }
}

int main() {
    int arr[] = {2, 6, 0, 1, 4};
    printArray(arr, 5);
}
```

Passing Arrays To Functions

What will this print?

```
1  // source/arrays_013.cpp
2  #include <iostream>
3  using namespace std;
4
5  void foo(int* arr, int length) {
6      for (int i = 0; i < length; i++) {
7          if (*(arr + i) == 0) {
8              *(arr + i) = 4;
9          }
10     }
11 }
12
13 int main() {
14     int arr[] = {2, 6, 0, 1, 4};
15     cout << arr[2] << endl;
16     foo(arr, 5);
17     cout << arr[2] << endl;
18 }
```

Array Decay

- A common C++ catch–phrase is that “arrays decay into pointers”.
- This simply means that, whenever required, arrays are interpreted as pointers, as we have already discussed above.
- As a consequence, when passing an array to a function, we are actually passing a pointer.
- This means that an array is always **passed by reference**. So, in case we need to pass an array be value, we have to devise various tricks we shall see in upcoming lectures.

Fun Time! (Optional)

Advanced Pointer Fun

While we have said enough about pointers, we have not explored pointer–land in full. The following tutorial will help you do so:

`https://learnmoderncpp.com/arrays-pointers-and-loops/`

Follow the tutorial step–by–step and pay attention to the “Experiments” it asks you to execute. Write down your observations in a document, which you will share with me at the end of the class at: `v.markos@mc-class.gr`.

Strings

Also, as a preparation for our next lecture – and an extension of the concepts we have explored in this lecture – follow the tutorial shown below:

`https://learnmoderncpp.com/string-and-character-literals/`

Again, pay attention to the detailed examples, making sure you understand what is going on there!

Homework

Complete all exercises and problems in MIT's C++ course second assignment, found here:

`https://ocw.mit.edu/courses/6-096-introduction-to-c-january-iap-2011/797ebff419fa2cc3a10af2c5f19be961_MIT6_096IAP11_assn02.pdf`

For your convenience, you can also find the assignment file in this lecture's materials, at: `../homework/MIT6-096IAP11-assn02.pdf`. Submit all your work in the online form below as a single `.zip` file:

`https://forms.gle/rSq3VSpcouRAVjqMA`

or via email at: `v.markos@mc-class.gr`.

Any Questions?

Do not forget to fill in the questionnaire shown right!

https://forms.gle/dKSrmE1VRVWqxBGZA