C++ Exercises
Set 5

Author(s): Huub Exel, Jessay Beukema, Sofia van der Wal, Ioannis Angelo Tassioulas

–

00:03

October 12, 2023

33

Description of differences between 1) pointers and arrays, 2) pointers and references, 3) how elements are reached using array indexing and pointers and 4) the definition of pointer arithmetic.

1. Pointer variables are datatypes which save the address of other variables. An array is an organized data structure which saves variables next to each other to be organized sequentially.
   The big difference in their implementation is their location in memory. Pointer variables can be stored anywhere in the memory, and simply contain the address of where the information can be found for the compiler to seek out. Arrays make sure to store the information of the variables which they keep right next to each other in memory. The name of the array defined is a pointer to the first position – when asked to index the array, for example array[0], you are asking for the data stored exactly 0 places/memory slots away from the beginning of the array.
   This way, parsing through an array means you must always go to the start and then walk through the entire array – with pointers, you can go directly to the data at the address you wish to arrive at. An array is a bit like a city bus, with predetermined stops you have to sit through to get around and a pointer is like a taxi, taking you exactly where you want to go.

2. Pointer variables are full variables, with a data value assigned and a location in memory given to it by the program. A reference variable is just an alias for another variable – they contain the same address data as the variable they are referencing, guiding the compiler to the same place. To the compiler, it is interpreted as a pointer which can never change what it is pointing to – a const pointer. A pointer, on the other hand, can be reassigned to different locations in memory.

3. In an array, first you must start at array[0][0], move down to array[3][0], then step twice to get to array[3][2]. With a pointer array, you simply call ∗pointer[3] to go straight to array[3][0], then you count two steps towards array[3][2].

4. Pointer arithmetic describes defining a separate pointer by either adding or subtracting from another pointer moves the memory location in which it points. For example, for a pointe ∗ptr pointing to memory location x, the pointer ∗(ptr + 1) points 1 step after the memory location of ∗ptr, which would be x + 1. They also follow the same commutivity laws of addition, which means that ∗(ptr + 1) = ∗(1 + ptr). Additionally, another important element of pointer arithmetic is the indirection operators & and ∗. '∗' removes a level of indirection by asking to reference the data of whatever it operates on. '&' Adds a level of indirection by asking to reference the address of the variable it operates on. & is treated as the mathematical inverse of ∗, and using one right after another leads to them canceling out to the identity (i.e. ∗&var = &∗var = var)

35

```
Create a CharCount class that counts all occurences of a character inside
of a file.
```

Listing 1: pub_header_charcount/charcount.hh

```
#ifndef INC1UDED_CHARCOUNT_
#define INC1UDED_CHARCOUNT_

#include <string>
#include <iostream>

class CharCount
{

    CharInfo d_charInfoStructObject;

    public:
        CharCount(CharInfo charInfoObject = CharInfo{});

        size_t count(std::istream &in = std::cin);
        CharInfo const &info() const;  // accessor d_charInfoStructObject

};

#endif
```

Listing 2: pub_header_charinfo/charinfo.hh

```
#ifndef INC1UDED_CHARINFO_
#define INC1UDED_CHARINFO_

struct CharInfo                 // This points to (struct) Char Objects
{
    Char *ptr;                  // Pointer to Char objects
    size_t nCharObj = 0;        // Number of Char objects pointed to
};

#endif
```

Listing 3: pub_header_char/char.hh

```
#ifndef INC1UDED_CHAR_
#define INC1UDED_CHAR_

#include <cstddef>

struct Char                 // Holds the amount of one distinct char in a file
{
    char ch;                // The distinct char that is counted
    size_t count = 0;       // The amount of times that char is in the file
};

#endif
```

Listing 4: declaration_showchar/showchar.hh

```
#ifndef INCLUDED_SHOWCHAR_
#define INCLUDED_SHOWCHAR_

void showChar(char const &ch);    // showChar function declaration

#endif
```

Listing 5: internal_header/charcount.ih

```
#include "../pub_header_char/char.hh"
```

```
#include "../pub_header_charinfo/charinfo.hh"
#include "../pub_header_charcount/charcount.hh"
#include "../declaration_showchar/showchar.hh"
// The order of these includes needs to be in this exact way

using namespace std;
```

Listing 6: constructor_charcount/constructor_charcount.cc

```
#include "../internal_header/charcount.ih"

CharCount::CharCount(CharInfo chInfoObj)
:                                       // Constructor that initializes the
    d_charInfoStructObject(chInfoObj)   // d_charInfoStructObject data member
{}                                      // with a default CharInfo object
```

Listing 7: count/count.cc

```
#include "../internal_header/charcount.ih"

size_t CharCount::count(istream &in)    // if no 'in' is given, this is cin
{
    string line;
    size_t amounOfChars;

    while (getline(in, line))           // Append the lenght of the lines to
        amounOfChars += line.length();  // the total

    return amounOfChars;                // Return amount of chars in the file
}
```

Listing 8: info/info.cc

```
#include "../internal_header/charcount.ih"

CharInfo const &CharCount::info() const
{
    return d_charInfoStructObject; // Returns the data member of type
}                                  // CharInfo (which is a struct)
```

36

```
Fill out the table with the pointer arithmetic notation and semantic meaning
of what each pointer is pointing to.

 ---------------------------------------------------------------------------
  definition:         rewrite:
 ---------------------------------------------------------------------------
  int x[8];           x[4]

pointer notation:  *(x + 4);
       semantics:  x + 4 points to the location of the 4th int beyond x.
                   That element is reached using the dereference operator (*)

 ---------------------------------------------------------------------------
  int x[8];           x[2] = x[3];

pointer notation:  *(x+2) = *(x+3);
       semantics:  set the data located in the second index after x to be
                   equal to the data located in the third index after x.
 ---------------------------------------------------------------------------
  char *argv[8];      cout << argv[2];

pointer notation:  cout << *(argv + 2);
       semantics:  Prints out the data of the 2nd index after argv[0],
                   which means that pointer data of argv[2] is printed.
 ---------------------------------------------------------------------------
```

```
  int x[8];           &x[10] - &x[3];

pointer notation:  x + 10 - x + 3;
       semantics:  subtract the location of the 10th index from the 3rd,
                   resulting in the distance between both address points.
```

```
  main's argv;        argv++[0];

pointer notation:  *argv++;
       semantics:  Prints out data at arg[0], then increments pointer to
                   point to next place in array - in this case, the beginning
                   of the next argument.
```

```
  main's argv;        argv[0]++;

pointer notation:  (*argv)++;
       semantics:  Increments the pointer to char of argv[0] by 1,
                   so next time argv[0] is called, the 0th char is
                   of the first argument is omitted (i.e. /a.o instead
                   of ./a.o)
```

```
  main's argv;        ++argv[0];

pointer notation:  ++(*argv[0]);
       semantics:  Increments the pointer to chars of first argument,
                   then returns the incremented value.
```

```
  main's argv;        ++argv[0][2];

pointer notation:  ++*(*argv + 2);
       semantics:  Increments the char, increasing its value. In the case
                   of argv being passed as "./a.o 1 2", argv[0][2] or
                   *(*argv + 2) refers to the 'a' char, incrementing to a 'b'
```

37

```
Translate written descriptions of pointers to C++ declarations.

#include <string>
int main(int argc, char **argv)
{
    double *example1 [8];                 // pointer to array of 8 doubles

    double **example2;                    // pointer to pointers to doubles

    std::string const **const *ptr;       // pointers to const pointers to
                                          // pointers to const strings

    std::string (*fun1()) [6][6];         // returns pointer to 6x6 array of
                                          // strings

    typedef std::string (*StringMat)[6];  // function fun returning a pointer
    StringMat (*fun2()) [6];              // to array of 6 StringMats, where
                                          // StringMats = arrays of 6 strings

    char** const fun(char* *const ptr);   // function receiving a variable
                                          // ptr, a const pointer to NTBSs,
                                          // returning a const NTBS

}
```

Listing 9: process.cc

```
#include <cstddef>
```

```
void process(size_t begin, size_t end, char const *const *args)
{
    for
    (
        const char *const *argsBegin = args + begin, *const *argsEnd = args +
           end;
            argsBegin != argsEnd;
                ++argsBegin
    )
        process(begin, end, argsBegin);
}
```