ETH zürich

Exercise Session W13

Computer Science (CSE & CBB & Statistics) – AS 23

Overview

Today's Agenda

Follow-up
Objectives
Pointers
Exercise "Push Back"
Memory Management
Outro

n.ethz.ch/~agavranovic

1. Follow-up

Follow-up from previous exercise sessions

- I messed up the Plan: The recap will be next week!
- Feel free to send me an email with your questions

2. Objectives

Objectives

- ☐ Understand the difference between `new` / `delete` and `new[]` / `delete[]`
- ☐ Be able to trace programs that use pointer arithmetic
- ☐ Be able to write programs that use pointer arithmetic

3. Pointers

`new` vs `new[]`

- `new T` allocates **one** space in memory for the specified type
- `new T[n]` allocates n spaces in memory for the specified type[1]
- Both return a pointer which points to the (first) element of the range

[1] this memory will be *contiguous*, i.e. "next to each other in memory"

Arrays

Statically allocated array

```
int myStatArr[3] = {2, 3, 8};
```

- `myStatArr` now points to the `2`
- `*myStatArr` returns `2`
- `myStatArr[2]` returns `8`
- `myStatArr[1] = -4` sets `3` to `-4`

Dynamically allocated arrays

```
int* myDynArr = new int[3]{2, 3, 8};
```

- `myDynArr` now points to the `2`
- `*myDynArr` returns `2`
- `myDynArr[2]` returns `8`
- `myDynArr[1] = -4` sets `3` to `-4`

`delete` vs `delete[]`

- We remember: every `new` needs a `delete`
- `delete[]` is the corresponding operator to `new[]`
- Be aware: We do not delete the pointer but the range of objects to which the pointer is pointing
- **Common source of bugs**
  Calling `delete` on the first element but not the entire array (with `delete[]`)

Pointer Arithmetic

- We can do "pointer math"
- The most important instructions are:

- Temporary shifts

```
ptr + 3
ptr - 3
```

- Permanent shifts

```
ptr++
--ptr
ptr += 2
```

- Determine distance between pointers

```
ptr_1 - ptr_2
```

- Compare positions

```
ptr_1 < ptr_2
ptr_1 != ptr_2
```

Questions?

Pointer Program

```
int* a = new int[5]{0, 8, 7, 2, -1};
int* ptr = a;                          // pointer assignment
++ptr;                                 // shift to the right
int my_int = *ptr;                     // read target
ptr += 2;                              // shift by 2 elements
*ptr = 18;                             // overwrite target
int* past = a+5;
std::cout << (ptr < past) << "\n"; // compare pointers
```

Pointer Program

Find and fix at least 3 problems in the following program.

```
#include <iostream>
int main () {
    int* a = new int[7]{0, 6, 5, 3, 2, 4, 1};
    int* b = new int[7];
    int* c = b;
    // copy a into b using pointers
    for (int* p = a; p <= a+7; ++p) {
        *c++ = *p;
    }
    // cross-check with random access
    for (int i = 0; i <= 7; ++i) {
        if (a[i] != c[i]) {
            std::cout << "Oops, copy error...\n";
        }
    }
    return 0;
}
```

Pointer Program

```
#include <iostream>
int main () {
    int* a = new int[7]{0, 6, 5, 3, 2,
    int* b = new int[7];
    int* c = b;
    // copy a into b using pointers
    for (int* p = a; p <= a+7; ++p) {
        *c++ = *p;
    }
    // cross-check with random access
    for (int i = 0; i <= 7; ++i) {
        if (a[i] != c[i]) {
            std::cout << "Oops, copy error...\n";
        }
    }
    return 0;
}
```

p = a+7 is dereferenced

Solution:
Use < instead of <=

Pointer Program

```
#include <iostream>
int main () {
    int* a = new int[7]{0, 6, 5, 3, 2, 4
    int* b = new int[7];
    int* c = b;
    // copy a into b using pointers
    for (int* p = a; p <= a+7; ++p) {
        *c++ = *p;
    }
    // cross-check with random access
    for (int i = 0; i <= 7; ++i) {
        if (a[i] != c[i]) {
            std::cout << "Oops, copy error
        }
    }
    return 0;
}
```

p = a+7 is dereferenced

Solution:
Use < instead of <=

Same problem as above

Pointer Program

```
#include <iostream>
int main () {
    int* a = new int[7]{0, 6, 5, 3, 2, 4, 1};
    int* b = new int[7];
    int* c = b;
    // copy a into b using pointers
    for (int* p = a; p <= a+7; ++p) {
        *c++ = *p;
    }
    // cross-check with random access
    for (int i = 0; i <= 7; ++i) {
        if (a[i] != c[i]) {
            std::cout << "Oops, copy err
        }
    }
    return 0;
}
```

`c` doesn't point to `b[0]` anymore.

Solution:
Use `b` instead of `c`

`p = a+7` is dereferenced

Solution:
Use `<` instead of `<=`

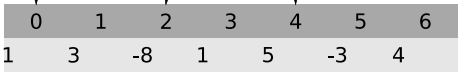
Same problem as above

Exercise – Applying Pointers

Exercise – Applying Pointers

```
// PRE: [b, e) and [o, o+(e-b)) are disjoint
//      valid ranges
void f (int* b, int* e, int* o) {
    while (b != e) {
        --e;
        *o = *e;
        ++o;
    }
}
```

Variable	Value
b	●
e	●
o	●



Exercise – Applying Pointers

Now determine a POST-condition for the function.

```
// PRE: [b, e) and [o, o+(e-b)) are disjoint
//      valid ranges
void f (int* b, int* e, int* o) {
    while (b != e) {
        --e;
        *o = *e;
        ++o;
    }
}
```

Exercise – Applying Pointers

```
// PRE: [b, e) and [o, o+(e-b)) are disjoint
//      valid ranges
// POST: The range [b, e) is copied in reverse
//       order into the range [o, o+(e-b))
void f (int* b, int* e, int* o) {
    while (b != e) {
        --e;
        *o = *e;
        ++o;
    }
}
```

Exercise – Valid Inputs

Exercise – Valid Inputs

- Which of these inputs are valid?

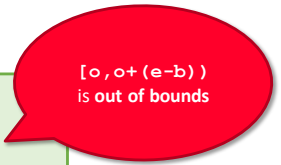
```
int* a = new int[5];
// Initialise a.
  a)  f(a, a+5, a+5);
  b)  f(a, a+2, a+3);
  c)  f(a, a+3, a+2);
```

```
// PRE: [b, e) and [o, o+(e-b)) are disjoint
//      valid ranges
void f (int* b, int* e, int* o) {
    while (b != e) {
        --e;
        *o = *e;
        ++o;
    }
}
```

Exercise – Valid Inputs

- Which of these inputs are valid?

```
int* a = new int[5];
// Initialise a.
  a)  f(a, a+5, a+5);
  b)  f(a, a+2, a+3);
  c)  f(a, a+3, a+2);
```




```
// PRE: [b, e) and [o, o+(e-b)) are disjoint
//      valid ranges
void f (int* b, int* e, int* o) {
    while (b != e) {
        --e;
        *o = *e;
        ++o;
    }
}
```

Exercise – Valid Inputs

- Which of these inputs are valid?

```
int* a = new int[5];
// Initialise a.
  a)  f(a, a+5, a+5);
  b)  f(a, a+2, a+3);
  c)  f(a, a+3, a+2);
```

[o,o+(e-b)) is out of bounds

```
// PRE: [b, e) and [o, o+(e-b)) are disjoint
//      valid ranges
void f (int* b, int* e, int* o) {
    while (b != e) {
        --e;
        *o = *e;
        ++o;
    }
}
```

Exercise – Valid Inputs

- Which of these inputs are valid?

```
int* a = new int[5];
// Initialise a.
  a)   f(a, a+5, a+5);
  b)   f(a, a+2, a+3);
  c)   f(a, a+3, a+2);
```

[o,o+(e-b)) is **out of bounds**

```
// PRE: [b, e) and [o, o+(e-b)) are disjoint
//      valid ranges
void f (int* b, int* e, int* o) {
    while (b != e) {
        --e;
        *o = *e;
        ++o;
    }
}
```

Ranges **not disjoint**

Questions?

Pointer Costness

There are two kinds of constness of pointers:

```
const int* ptr = &a;
```

no write-access to target **a**
i.e. we are *not* allowed to change the value of the integer **a**

```
int* const ptr = &a;
```

no write-access to pointer **ptr**
i.e. we are not allowed to change to where the pointer points

Exercise – `const` Correctness

Exercise – `const` Correctness

- Make the function `const`-correct.

```
// PRE: [b, e) and [o, o+(e-b)) are disjoint
//      valid ranges
void f (int* b, int* e, int* o) {
    while (b != e) {
        --e;
        *o = *e;
        ++o;
    }
}
```

(From: Script Exercise 113)

Exercise – `const` Correctness

- Make the function `const`-correct.

```
// PRE: [b, e) and [o, o+(e-b)) are disjoint
//      valid ranges
void f (const int* const b, const int* e, int* o) {
    while (b != e) {
        --e;
        *o = *e;
        ++o;
    }
}
```

Questions?

4. Exercise "Push Back"

Exercise "Push Back"

- Open "Push Back" on **code** expert
- Think about how you would approach the problem with pen and paper
- Implement a solution (optionally in groups)

Solution "Push Back"

```
// PRE: source_begin points to first element to be copied.
// PRE: source_ends points to element after the last element to be copied.
// PRE: destination_begin points to first element of target memory block
// PRE: #elements in target memory location >= #elements in source
// POST: copies the content of the source memory block to the destination
//       memory block.
void copy_range(const int* const source_begin,
                const int* const source_end,
                int* const destination_begin   ){

  int* dst = destination_begin;
  for (const int* src = source_begin; src != source_end; ++src) {
    *dst = *src;
    ++dst;
  }
}
```

Solution "Push Back"

```
void our_vector::push_back(int new_element) {
  // 1. Allocate a new memory block larger by one element
  unsigned int lenghtOfNewBlock = this->count + 1;
  int* const ptrToNewBlock = new int[lenghtOfNewBlock];

  // 2. Copy all the elements from the old memory block to the new one
  copy_range(this->elements, this->elements + count, ptrToNewBlock);

  // 3. Deallocate the old memory block
  delete[] this->elements;          // frees memory from old elements
  this->elements = ptrToNewBlock;   // redirects pointer to new block

  // 4. Add the new element at the end of the new memory block
  this->elements[count] = new_element;
  count++;                          // increment counter
}
```

Questions?

5. Memory Management

Find mistakes in the following code and suggest fixes:

```
1  // PRE: len is the length of the memory block that starts at array
2  void test1(int* array, int len) {
3      int* fourth = array + 3;
4      if (len > 3) {
5          std::cout << *fourth << std::endl;
6      }
7      for (int* p = array; p != array + len; ++p) {
8          std::cout << *p << std::endl;
9      }
10 }
```

Find mistakes in the following code and suggest fixes:

```
1  // PRE: len is the length of the memory block that starts at array
2  void test1(int* array, int len) {
3      //int* fourth = array + 3;       // ERROR
4      if (len > 3) {
5          int* fourth = array + 3;     // OK
6          std::cout << *fourth << std::endl;
7      }
8      for (int* p = array; p != array + len; ++p) {
9          std::cout << *p << std::endl;
10     }
11 }
```

Even if the pointer is not dereferenced, it must point into a memory block or to the element just after its end.

Find mistakes in the following code and suggest fixes:

```
1  // PRE: len >= 2
2  int* fib(unsigned int len) {
3      int* array = new int[len];
4      array[0] = 0; array[1] = 1;
5      for (int* p = array+2; p < array + len; ++p) {
6          *p = *(p-2) + *(p-1); }
7      return array; }
8  void print(int* array, int len) {
9      for (int* p = array+2; p < array + len; ++p) {
10         std::cout << *p << " ";
11     }
12 }
13 void test2(unsigned int len) {
14     int* array = fib(len);
15     print(array, len);
16 }
```

```
1  // PRE: len >= 2
2  int* fib(unsigned int len) {
3      int* array = new int[len];
4      array[0] = 0; array[1] = 1;
5      for (int* p = array+2; p < array + len; ++p) {
6          *p = *(p-2) + *(p-1); }
7      return array; }
8  void print(int* array, int len) {
9      for (int* p = array+2; p < array + len; ++p) {
10         std::cout << *p << " ";
11     }
12 }
13 void test2(unsigned int len) {
14     int* array = fib(len);
15     print(array, len);
16 }   // array is leaked; to fix add: delete[] array
```

Find mistakes in the following code and suggest fixes:

```
1  // PRE: len >= 2
2  int* fib(unsigned int len) {
3      // ...
4  }
5  void print(int* m, int len) {
6      for (int* p = m+2; p < m + len; ++p) {
7          std::cout << *p << " ";
8      }
9      delete m;
10 }
11 void test2(unsigned int len) {
12     int* array = fib(len);
13     print(array, len);
14     delete[] array;
15 }
```

```
1  // PRE: len >= 2
2  int* fib(unsigned int len) {
3      // ...
4  }
5  void print(int* m, int len) {
6      for (int* p = m+2; p < m + len; ++p) {
7          std::cout << *p << " ";
8      }
9      delete m;   // should be delete[]
10 }
11 void test2(unsigned int len) {
12     int* array = fib(len);
13     print(array, len);
14     delete[] array; // array deallocated twice
15 }
```

Questions?

6. Outro

General Questions?

Till next time!

Cheers!