Exercise Session — Computer Science — 12

Pointer Arithmetic, Memory Management

Overview

Today's Plan

Pointers
- Example: Pointers on Arrays
- Example: Special Copy

Exercise "Push Back"
Memory Management

n.ethz.ch/~iopopa

Link to Webpage

Send an e-Mail

Follow-up from last session

- **Can we manually delete memory pointed to by a Smart Pointer?**
- No, because smart pointers automatically manage the memory they own and delete it when they go out of scope or when the reference count drops to zero.

Follow-up from last session

- **Can we manually delete memory pointed to by a Smart Pointer?**
- No, because smart pointers automatically manage the memory they own and delete it when they go out of scope or when the reference count drops to zero.
- If you manually delete memory managed by a smart pointer, the smart pointer will attempt to delete the same memory again when it goes out of scope, resulting in undefined behavior.

Follow-up from last session

Main disadvantages of smart pointers

Follow-up from last session

Main disadvantages of smart pointers

- **Compatibility Issues**
  - Smart pointers might not always be compatible with libraries that expect raw pointers or use their own memory management schemes.
  - Converting between smart pointers and raw pointers (.get()) can introduce risks if not handled properly.

Follow-up from last session

Main disadvantages of smart pointers

- **Compatibility Issues**
  - Smart pointers might not always be compatible with libraries that expect raw pointers or use their own memory management schemes.
  - Converting between smart pointers and raw pointers (.get()) can introduce risks if not handled properly.
- **Unnecessary Overhead for Simple Cases**

Follow-up from last session

Main disadvantages of smart pointers

- **Compatibility Issues**
  - Smart pointers might not always be compatible with libraries that expect raw pointers or use their own memory management schemes.
  - Converting between smart pointers and raw pointers (.get()) can introduce risks if not handled properly.
- **Unnecessary Overhead for Simple Cases**
- **Performance Overhead**

1. Pointers

`new` vs `new[]`

- `new T` allocates **one** space in memory for the specified type

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

`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]

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

`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

Arrays

Statically allocated array

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

- `myStatArr` now points to the

Arrays

Statically allocated array

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

- `myStatArr` now points to the 2
- `*myStatArr` returns

Arrays

Statically allocated array

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

- `myStatArr` now points to the 2
- `*myStatArr` returns 2
- `myStatArr[2]` returns

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`

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

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

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};
```

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

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

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

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`

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

But what is the difference between them?

Statically allocated array:

- Memory is allocated at compile time on the stack.
- The size of the array must be known at compile time and cannot be changed during runtime.

Dynamically allocated arrays:

- Memory is allocated at runtime on the heap using new.
- The size can be specified during runtime, allowing for more flexibility.

`delete` vs `delete[]`

- We remember:

`delete` vs `delete[]`

- We remember: every **`new`** needs a **`delete`**

`delete` vs `delete[]`

- We remember: every **`new`** needs a **`delete`**
- **`delete`**`[]` is the corresponding operator to **`new`**`[]`

`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

`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:

Pointer Arithmetic

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

- Temporary shifts

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

Pointer Arithmetic

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

- Temporary shifts

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

- Permanent shifts

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

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
```

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?

1. Pointers

1.1. Example: Pointers on Arrays

Pointer Arithmetic

```cpp
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
  // ^ Note how this does not simply "add 2" to the
  // underlying memory address, but instead adds the
  // appropriate amount to get to the integer variable
  // that is stored "2 ints further away"
*ptr = 18;                                       // overwrite target
int* past = a+5;
std::cout << (ptr < past) << "\n";               // compare pointers
```

Bug Hunt

Find and fix at least 3 problems in the following program

```
int* a = new int[7]{0, 6, 5, 3, 2, 4, 1};
int* b = new int[7];
int* c = b;

for (int* p = a; p <= a+7; ++p) {   // copy a into b using pointers
    *c++ = *p;
}

for (int i = 0; i <= 7; ++i) {      // cross-check with random access
    if (a[i] != c[i]) {
        std::cout << "Oops, copy error...\n";
    }
}
```

Bug Hunt

Find and fix at least 3 problems in the following program

```
int* a = new int[7]{0, 6, 5, 3, 2, 4, 1};
int* b = new int[7];
int* c = b;

for (int* p = a; p <= a+7; ++p) {   // copy a into b using pointers
    *c++ = *p;
}

for (int i = 0; i <= 7; ++i) {       // cross-check with random access
    if (a[i] != c[i]) {
        std::cout << "Oops, copy error...\n";
    }
}
```

Problems: `p`, `i` are dereferenced at `a+7`; `c` doesn't point to `b[0]` anymore!

Questions?

1.2. Example: Special Copy

Special Copy?

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

Reverse Copy!

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

Reverse Copy!

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

Which of these inputs are valid after `int* a = new int[5];`?

a) `f(a, a+5, a+5)` b) `f(a, a+2, a+3)` c) `f(a, a+3, a+2)`

Reverse Copy!

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

Which of these inputs are valid after `int* a = new int[5];`?

a) `f(a, a+5, a+5)` b) `f(a, a+2, a+3)` c) `f(a, a+3, a+2)`

Answer: b)

Questions?

Pointer Costness

There are two kinds of constness of pointers:

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

Pointer Costness

There are two kinds of constness of pointers:

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

no write-access to `a`

Pointer Costness

There are two kinds of constness of pointers:

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

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

Pointer Costness

There are two kinds of constness of pointers:

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

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

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

Pointer Costness

There are two kinds of constness of pointers:

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

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

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

no write-access to `ptr`

Pointer Costness

There are two kinds of constness of pointers:

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

no write-access to `a`

i.e. we are *not* allowed to change the value of the integer `a`

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

no write-access to `ptr`

i.e. we are not allowed to change to where the pointer points

Questions?

2. Exercise "Push Back"

Exercise "Push Back"

- Open "Push Back" on **code** expert

Exercise "Push Back"

- Open "Push Back" on **code** expert
- Think about how you would approach the problem with pen and paper

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;
//       source_ends points to element after the last element to be copied;
//       destination_begin points to first element of target memory block;
//       #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?

3. Memory Management

Bug Hunt I

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

Find mistakes in the code and suggest fixes

Bug Hunt I — Dangerous Pointer

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

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

Bug Hunt II

```
// PRE: len >= 2
int* fib(int len) {
    int* array = new int[len];
    array[0] = 0; array[1] = 1;
    for (int* p = array+2; p < array + len; ++p) {
        *p = *(p-2) + *(p-1); }
    return array; }
void print(int* array, int len) {
    for (int* p = array+2; p < array + len; ++p) {
        std::cout << *p << " ";
    }
}
void test2(int len) {
    int* array = fib(len);
    print(array, len);
}
```

Bug Hunt II — Memory Leak

```
// PRE: len >= 2
int* fib(int len) {
    int* array = new int[len];
    array[0] = 0; array[1] = 1;
    for (int* p = array+2; p < array + len; ++p) {
        *p = *(p-2) + *(p-1); }
    return array; }
void print(int* array, int len) {
    for (int* p = array+2; p < array + len; ++p) {
        std::cout << *p << " ";
    }
}
void test2(int len) {
    int* array = fib(len);
    print(array, len);
    delete[] array;         // otherwise array is leaked!
}
```

Bug Hunt III

```
// PRE: len >= 2
int* fib(int len) {
    // ...
}
void print(int* m, int len) {
    for (int* p = m+2; p < m + len; ++p) {
        std::cout << *p << " ";
    }
    delete m;
}
void test2(int len) {
    int* array = fib(len);
    print(array, len);
    delete[] array;
}
```

Bug Hunt III — Double Free!

```
// PRE: len >= 2
int* fib(int len) {
    // ...
}
void print(int* m, int len) {
    for (int* p = m+2; p < m + len; ++p) {
        std::cout << *p << " ";
    }
    delete[] m;
}
void test2(int len) {
    int* array = fib(len);
    print(array, len);
    // delete[] array;            // array deallocated twice!
}
```

Questions?

4. Outro

General Questions?

See you next time

Have a nice week!