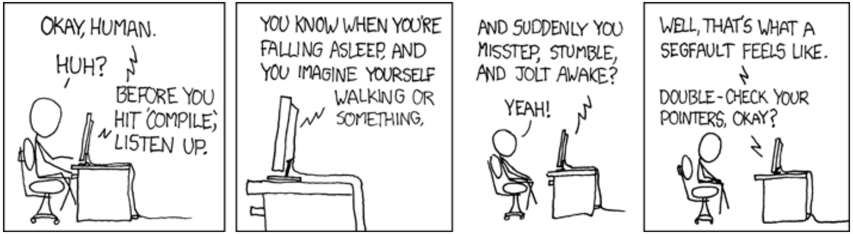
Informatik - Exercise Session


OKAY, HUMAN.
HUH?
BEFORE YOU HIT 'COMPILE,' LISTEN UP.
YOU KNOW WHEN YOU'RE FALLING ASLEEP, AND YOU IMAGINE YOURSELF WALKING OR SOMETHING,
AND SUDDENLY YOU MISSTEP, STUMBLE, AND JOLT AWAKE?
YEAH!
WELL, THAT'S WHAT A SEGFAULT FEELS LIKE.
DOUBLE-CHECK YOUR POINTERS, OKAY?

`https://xkcd.com/371/`

Pointer error example

Find mistakes in the following code and suggest fixes:

```
// 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;
    }
}
```

Pointer error example

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 }
```

Line 3 produces an error: Even if the pointer is not dereferenced, it must point into a memory block or to the element just after its end. To fix this, move line 3 inside the `if` branch.

Array memory error example

Find mistakes in the following code and suggest fixes:

```cpp
// 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);
}
```

Array memory error example

Find mistakes in the following code and suggest fixes:

```
// 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);
}
```

array is leaked, to fix this, add a `delete[] array` somewhere.

Array memory error example reloaded

Find mistakes in the following code and suggest fixes:

```cpp
// 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 memory error example reloaded

Find mistakes in the following code and suggest fixes:

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

On line 5, it should be `delete[]`.

Array memory error example reloaded

Find mistakes in the following code and suggest fixes:

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

On line 5, it should be `delete[]`. On line 10, we free the memory of `array` a second time. To fix this, remove one of the `delete[]`'s.

Memory ownership

The main point of the last two examples is to show you that correctly allocating and deallocating memory is a non-trivial task. This is especially a problem in large code bases where objects are often deallocated far away from the location in which they were created. Therefore, when dealing with dynamic memory it is useful to think in terms of *ownership*. The core idea of ownership is that a memory location can have only *one owner at a time* and that owner is responsible for deallocating it.

Memory ownership

The main point of the last two examples is to show you that correctly allocating and deallocating memory is a non-trivial task. This is especially a problem in large code bases where objects are often deallocated far away from the location in which they were created. Therefore, when dealing with dynamic memory it is useful to think in terms of *ownership*. The core idea of ownership is that a memory location can have only *one owner at a time* and that owner is responsible for deallocating it.

Example: The function `fib` allocates an array and transfers the ownership of that array to the caller. When `test2` calls `print`, it transfers the ownership of the array to it. When `print` finishes its work, we have a choice: either `print` deallocates the array or returns its ownership to the caller. In the latter case, `test2` is responsible for deallocating the array.

Memory ownership

The main point of the last two examples is to show you that correctly allocating and deallocating memory is a non-trivial task. This is especially a problem in large code bases where objects are often deallocated far away from the location in which they were created. Therefore, when dealing with dynamic memory it is useful to think in terms of *ownership*. The core idea of ownership is that a memory location can have only *one owner at a time* and that owner is responsible for deallocating it.

Example: The function `fib` allocates an array and transfers the ownership of that array to the caller. When `test2` calls `print`, it transfers the ownership of the array to it. When `print` finishes its work, we have a choice: either `print` deallocates the array or returns its ownership to the caller. In the latter case, `test2` is responsible for deallocating the array.

Additionally: Memory can be owned not only by *functions*, but also by *classes*. For example, the stack shown in the lecture owns the dynamically allocated memory. Therefore, we should never deallocate that memory from outside of the stack.