Exercise 9 – Pointers, Iterators and Recursion

Informatik I für Mathematiker und Physiker (HS 2015)

Yeara Kozlov

Slides courtesy of Virag Varga

Agenda

- Passing arrays to functions
- Iterators
- const iterators and pointers
- Recursion

Parameter by value / pointer /reference

```
#include <iostream>


void foo (int a, int* b, int& c) {
  a = 10;
  *b = 20;
  c = 30;
}

int main() {
  int a = 0, b = 0, c = 0;
  foo(a, &b, c);
  std::cout << a << " "<< b << " "<< c << "\n";       // ?

  return 0;
}
```

Arrays and pointers

```
int my_array[5] = {1,2,3,4,5};      // static array
int* p;                             // pointer to int
p = &my_array[0];                   // pointer to 0th element
p = my_array;                       // pointer to 0th element
```

- `my_array` and `&my_array[0]` are the same
  - The address of the 0th element in the array

```
int cards[5];
```

```
cards[0] ←→ *cards      ←→ the value at address cards
cards[3] ←→ *(cards+3)  ←→ the value at address (cards+3)
```

Passing array as a pointer

```
#include <iostream>


// PRE: [begin, end] is a valid range
// POST: added c to each element
void add_to_elements (int* begin, int* end, const int c) {
  while (begin < end) {
    *begin += c;
    ++begin;
  }
}

int main() {
  int arr[5] = {0, 1, 2, 3, 4};
  add_to_elements(arr, arr+5, 10);


  int arr2[5] = {0, 1, 2, 3, 4};
  add_to_elements(arr2+1, arr2+3, 10);
  return 0;
}
```

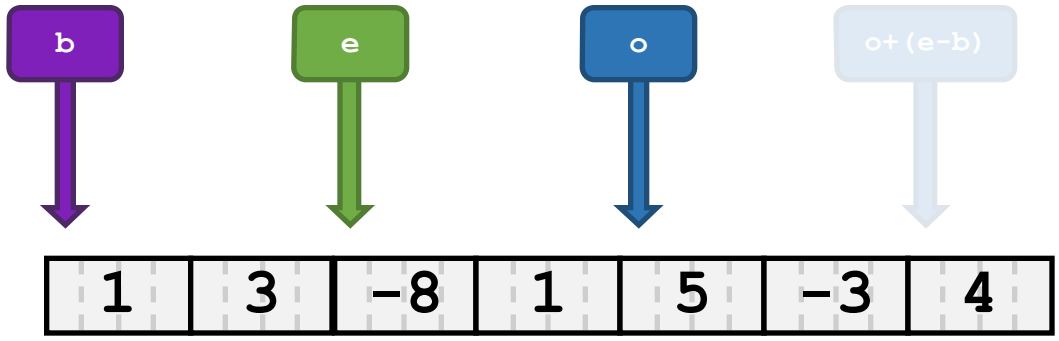
Exercise – Applying Pointers

Exercise – Applying Pointers

- Apply this 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;
    }
}
```

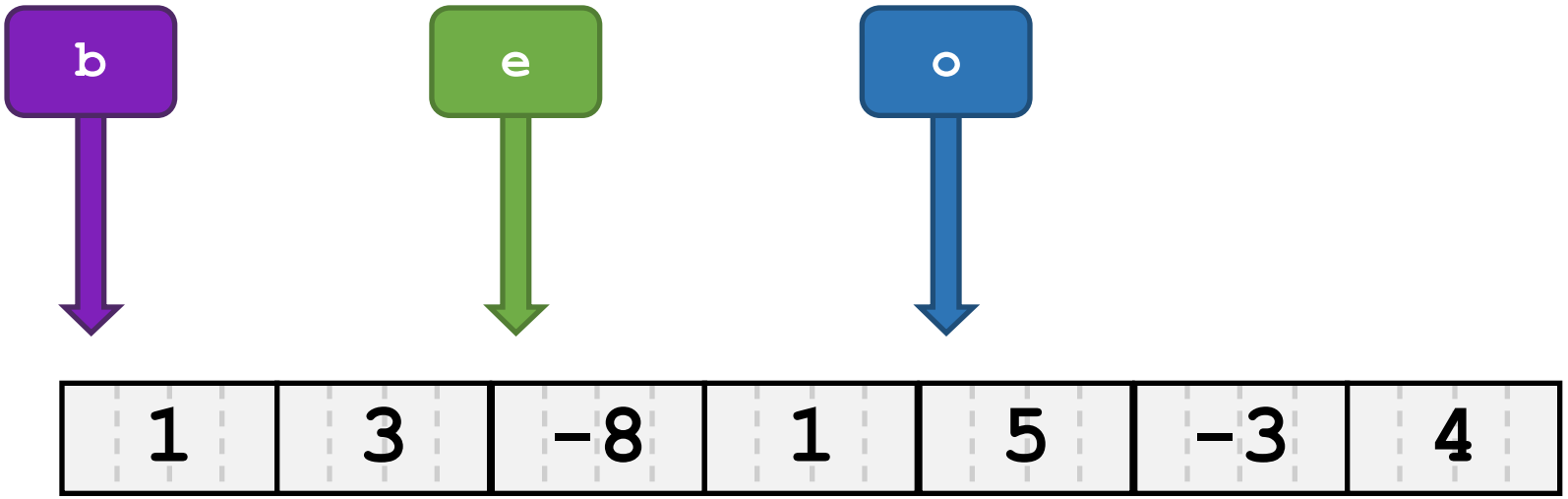
- … to this example-array:



(From: Script Exercise 113)

Exercise – Applying Pointers

```
void f (int* b, int* e, int* o) {
    while (b != e) {
        --e;
        *o = *e;
        ++o;
    }
}
```

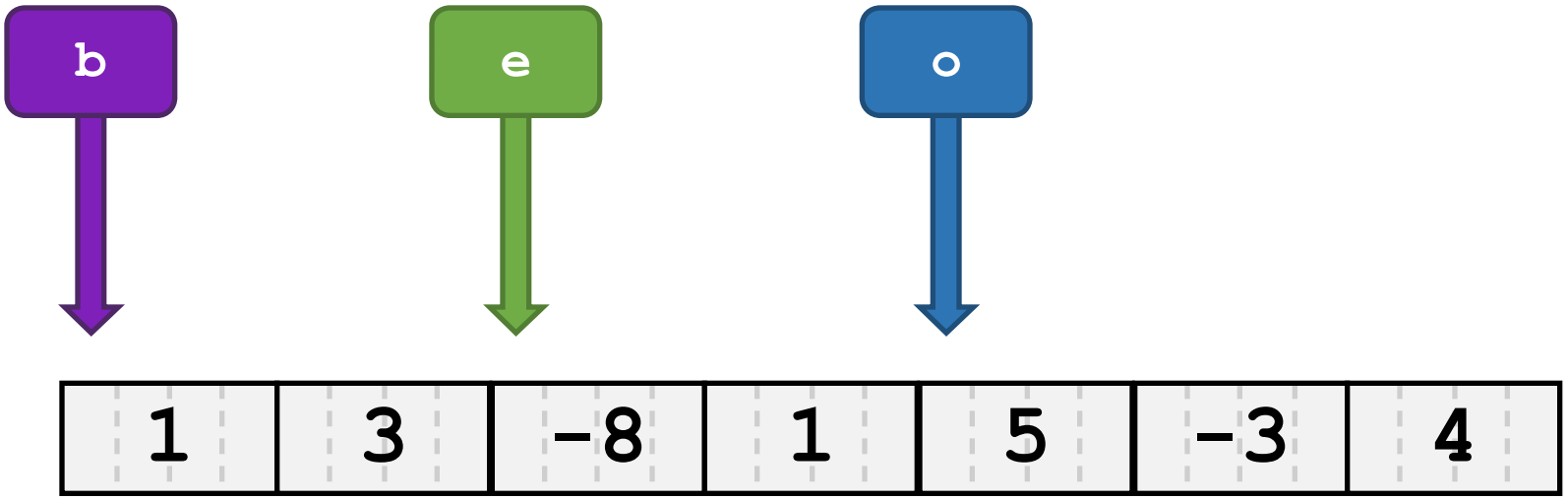


(From: Script Exercise 113)

Exercise – Applying Pointers

true

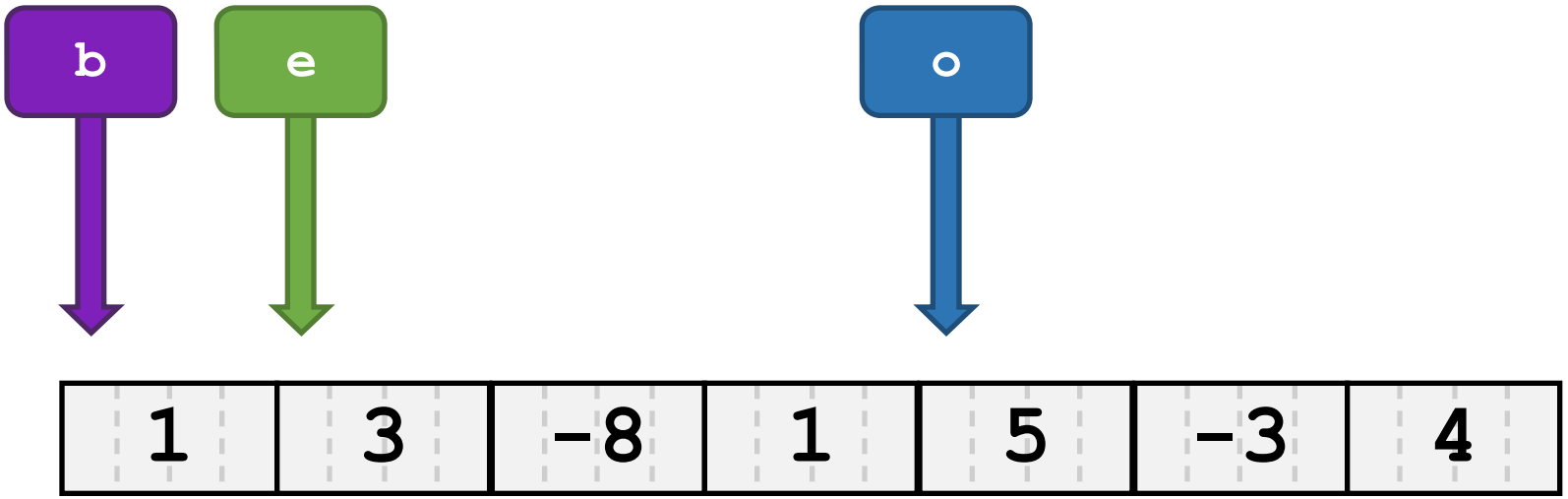
```
void f (int* b, int* e, int* o) {
    while (b != e) {
        --e;
        *o = *e;
        ++o;
    }
}
```



(From: Script Exercise 113)

Exercise – Applying Pointers

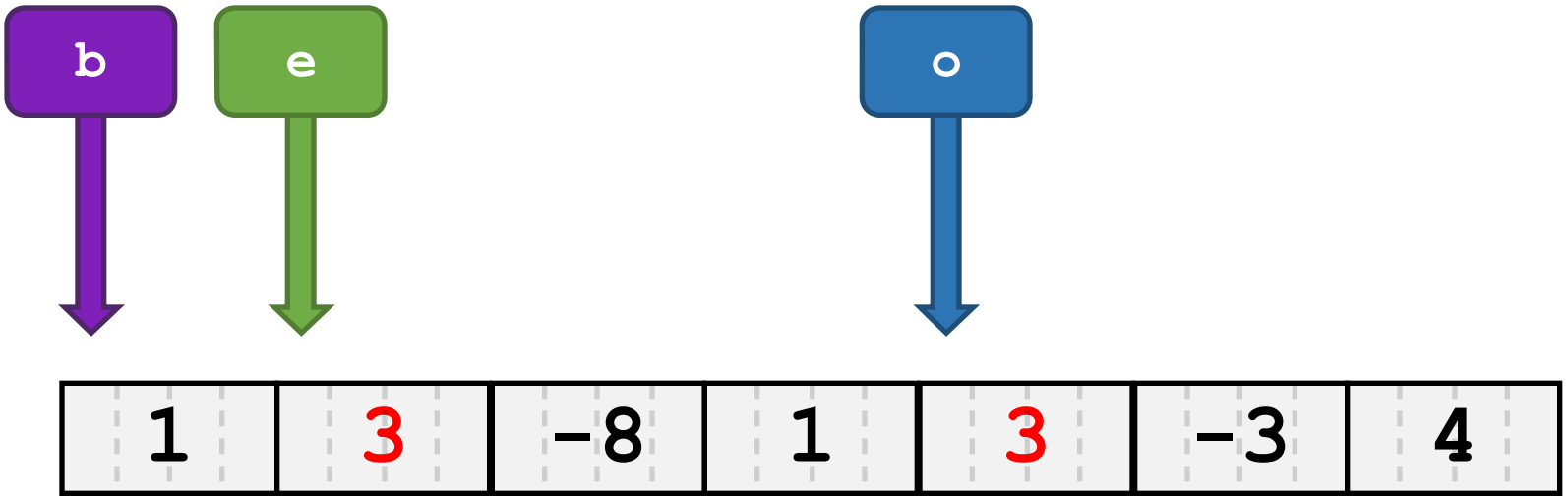
```
void f (int* b, int* e, int* o) {
    while (b != e) {
        --e;
        *o = *e;
        ++o;
    }
}
```



(From: Script Exercise 113)

Exercise – Applying Pointers

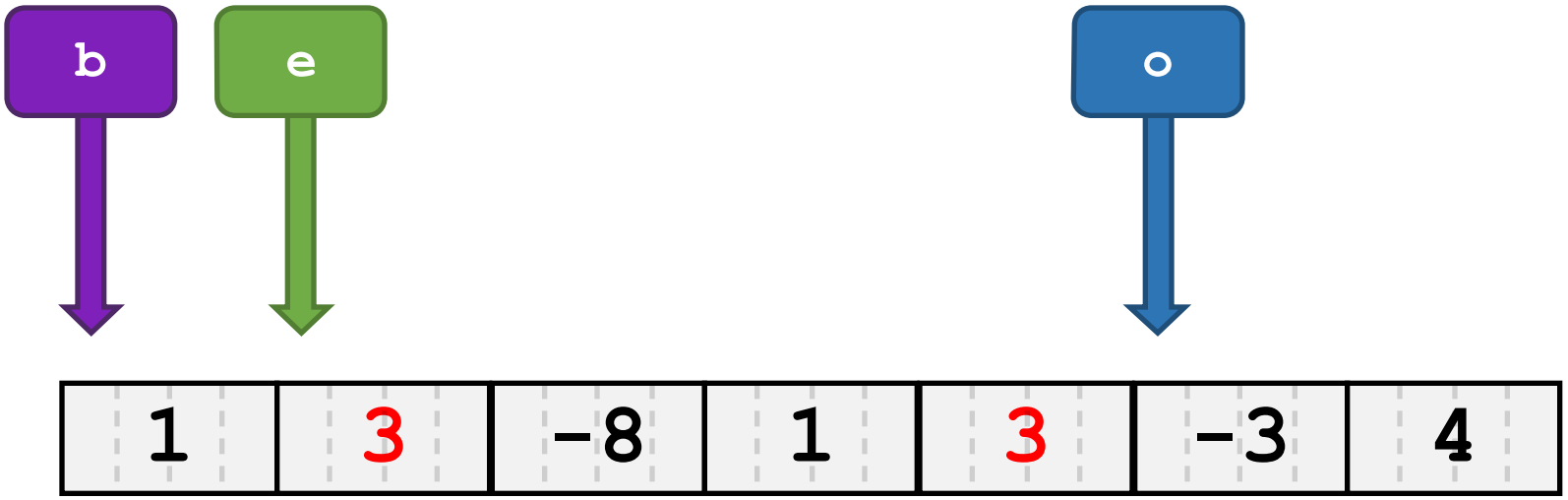
```
void f (int* b, int* e, int* o) {
    while (b != e) {
        --e;
        *o = *e;
        ++o;
    }
}
```



(From: Script Exercise 113)

Exercise – Applying Pointers

```
void f (int* b, int* e, int* o) {
    while (b != e) {
        --e;
        *o = *e;
        ++o;
    }
}
```

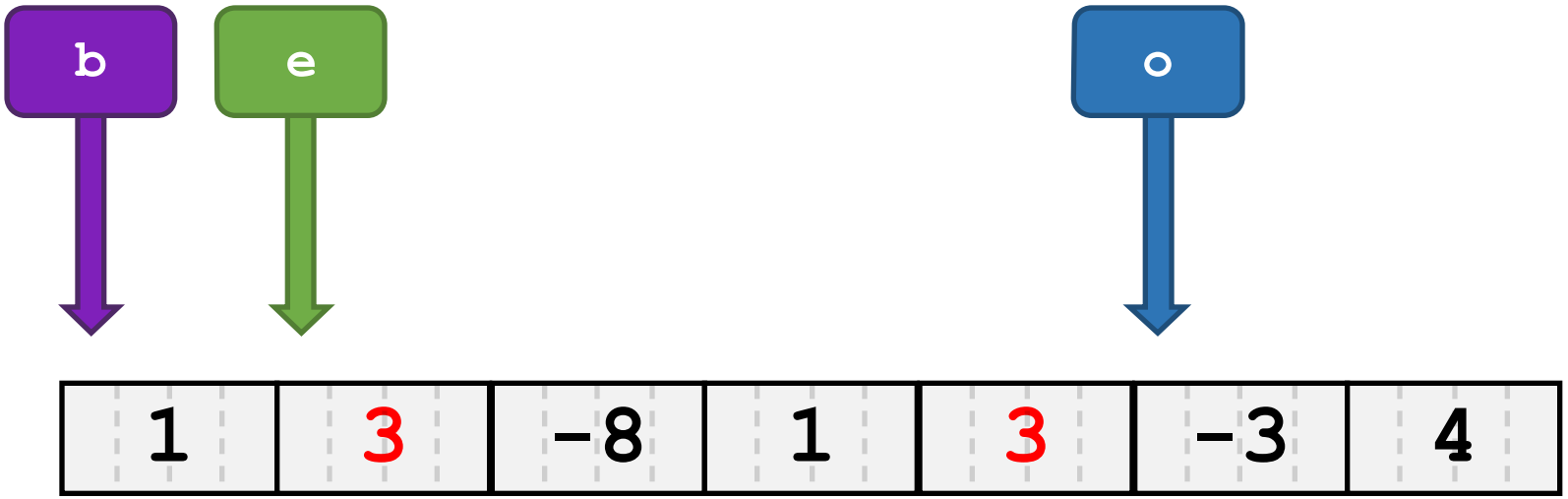


(From: Script Exercise 113)

Exercise – Applying Pointers

true

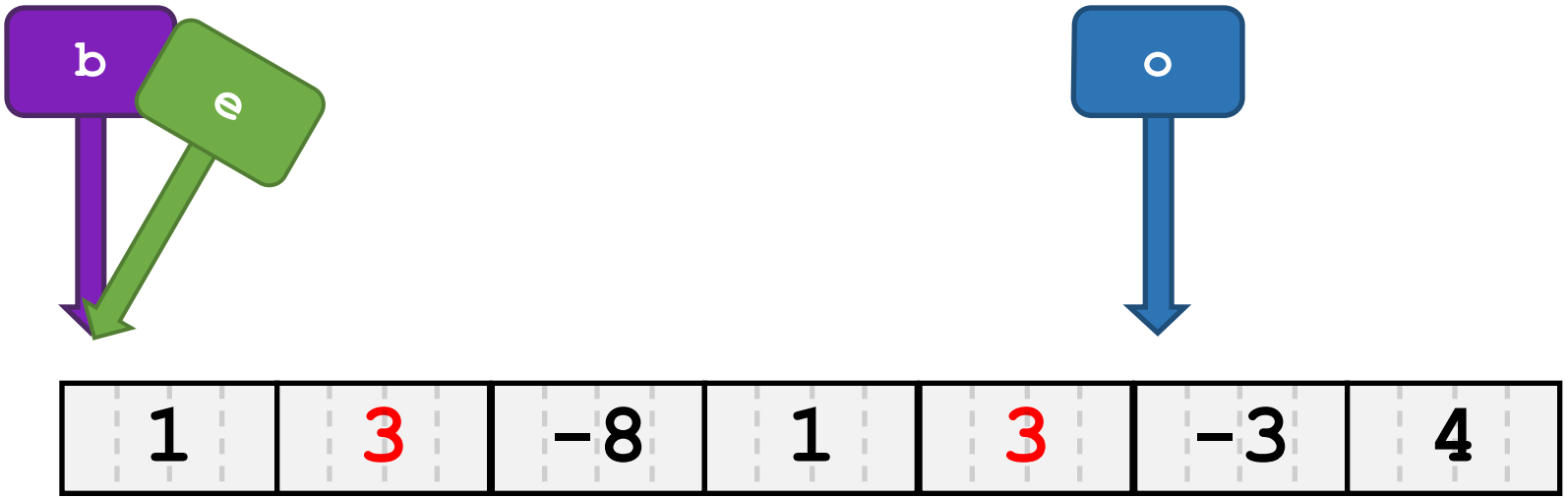
```
void f (int* b, int* e, int* o) {
    while (b != e) {
        --e;
        *o = *e;
        ++o;
    }
}
```



(From: Script Exercise 113)

Exercise – Applying Pointers

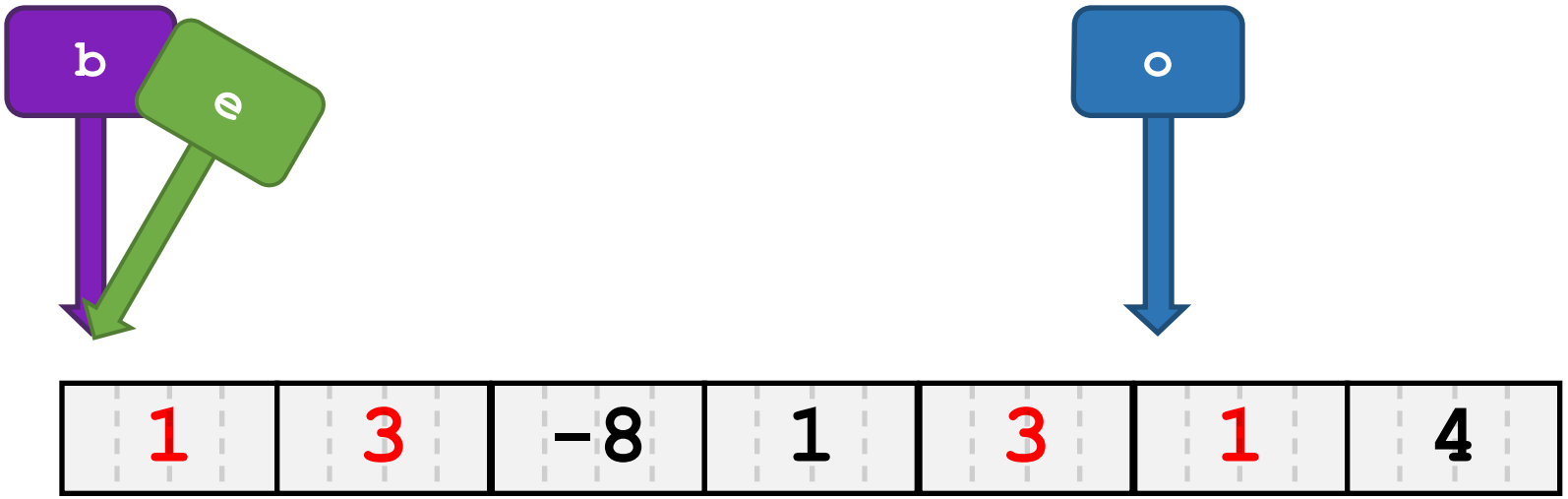
```
void f (int* b, int* e, int* o) {
    while (b != e) {
        --e;
        *o = *e;
        ++o;
    }
}
```



(From: Script Exercise 113)

Exercise – Applying Pointers

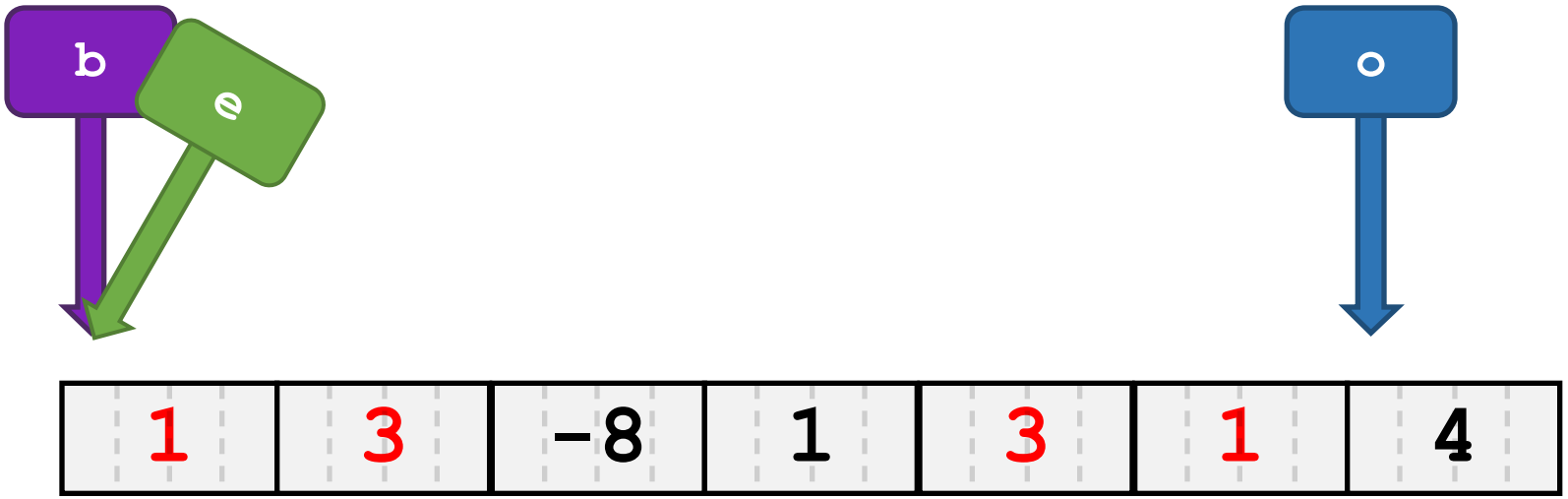
```
void f (int* b, int* e, int* o) {
    while (b != e) {
        --e;
        *o = *e;
        ++o;
    }
}
```



(From: Script Exercise 113)

Exercise – Applying Pointers

```
void f (int* b, int* e, int* o) {
    while (b != e) {
        --e;
        *o = *e;
        ++o;
    }
}
```

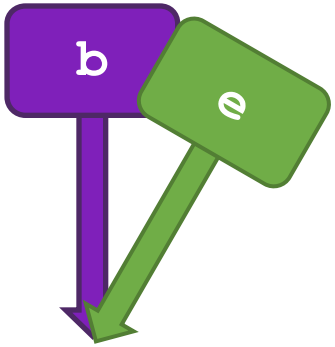


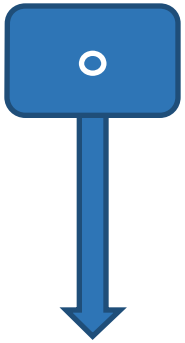
(From: Script Exercise 113)

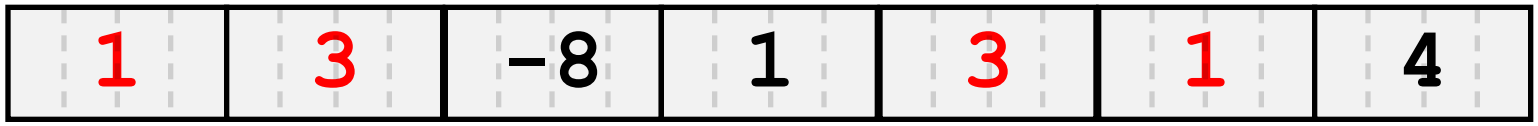
Exercise – Applying Pointers

false

```
void f (int* b, int* e, int* o) {
    while (b != e) {
        --e;
        *o = *e;
        ++o;
    }
}
```







(From: Script Exercise 113)

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;
    }
}
```


(From: Script Exercise 113)

Exercise – Applying Pointers

- Something like this:

```
// 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;
    }
}
```


(From: Script Exercise 113)

Exercise – Valid Inputs

- Which of these inputs are valid?

```
int a[5] = {1, 2, 3, 4, 5};
  a)  f(a, a+5, a+5);
  b)  f(a, a+2, a+3);
  c)  f(a, a+3, a+2);
```

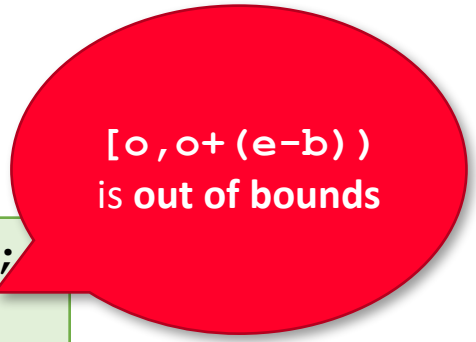
```
// PRE: [b, e) and [o, o+(e-b)) are disjoint
//      valid ranges
void f (int* b, int* e, int* o) {
    while (b != e) {
        --e;
        *o = *e;
        ++o;
    }
}
```


(From: Script Exercise 113)

Exercise – Valid Inputs

- Which of these inputs are valid?

```
int a[5] = {1, 2, 3, 4, 5};
  a)  f(a, a+5, a+5);
  b)  f(a, a+2, a+3);
  c)  f(a, a+3, a+2);
```



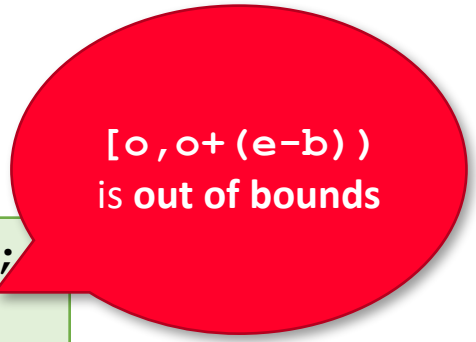

```
// PRE: [b, e) and [o, o+(e-b)) are disjoint
//      valid ranges
void f (int* b, int* e, int* o) {
    while (b != e) {
        --e;
        *o = *e;
        ++o;
    }
}
```


(From: Script Exercise 113)

Exercise – Valid Inputs

- Which of these inputs are valid?

```
int a[5] = {1, 2, 3, 4, 5};
  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;
    }
}
```


(From: Script Exercise 113)

Exercise – Valid Inputs

- Which of these inputs are valid?

```
int a[5] = {1, 2, 3, 4, 5};
  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**

Ranges **not disjoint**

```
// 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)

Arrays : pointers ⇔ vectors : iterators

- Looping through an array with pointers:

```
int a[5] = {0,1,2,3,4};                    // static array
for(int* p = &a[0]; p < &a[0] + 5; ++p){
  std::cout << *p;
}
```

- Looping through a vector with iterators

```
std::vector<int> v[5] = {0,1,2,3,4};     // std vector
for(std::vector<int>::iterator it= v.begin(); it < v.end(); ++it){
   std::cout << *it;
}
```

Passing array as a pointer

```
#include <iostream>


// PRE: [begin, end] is a valid range
// POST: added c to each element
void add_to_elements (int* begin, int* end, const int c) {
  while (begin < end) {
    *begin += c;
    ++begin;
  }
}

int main() {
  int arr[5] = {0, 1, 2, 3, 4};
  add_to_elements(arr, arr+5, 10);


  int arr2[5] = {0, 1, 2, 3, 4};
  add_to_elements(arr2+1, arr2+3, 10);
  return 0;
}
```

Passing vector as an iterator

```
#include <iostream>
#include <vector>
// PRE: [begin, end] is a valid range
// POST: added c to each element
void add_to_elements (std::vector<int>::iterator begin,
                      std::vector<int>::iterator end, const int c) {
  while (begin < end) {
    *begin += c;
    ++begin;
  }
}

int main() {
  std::vector<int> vec = {0, 1, 2, 3, 4};
  add_to_elements(vec.begin(), vec.end(), 10);


  std::vector<int> vec2 = {0, 1, 2, 3, 4};
  add_to_elements(vec2.begin()+1, vec2.begin()+3, 10);
  return 0;
}
```

typedef

- Iterators are great, but function definitions become very long. We would like to define shorter type names:

```
std::vector<int> vec;                    →    int_vec vec;

std::vector<int>::iterator it;           →    int_vec_it it;
```

- Using the keyword **typedef**:

```
typedef std::vector<int> int_vec;
typedef std::vector<int>::iterator int_vec_it;


int_vec vec = {0, 1, 2, 3, 4};
for(int_vec_it it= vec.begin(); it < vec.end(); ++it){
  std::cout << *it;
}
```

typedef hints

- Code is easier to read.
- Code is easier to maintain.
- You can change the precision for multiple variables by changing one line of code:
  - `typedef float ftype;`
  - `typedef double ftype;`

const pointers

- Pointers and references prevent unnecessary data duplication, but give functions ability to modify the memory.
- The const keyword gives write-protection through pointer / reference access

```
int x = 7;
int y = 5;


const int* i = &x;          // ?
++(*i);                     // ?
i = &y;                     // ?


const int& j = x;           // ?
++j;                        // ?
j = y;                      // ?
```

const iterators

- Write-protection for vectors

```
std::vector <int> x(5, 0);


std::vector <int>::const_iterator i = x.begin();
++(*i);                          // ?
i = x.begin() + 3;               // ?
```

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* b, const int* e, int* o) {
    while (b != e) {
        --e;
        *o = *e;
        ++o;
    }
}
```


(From: Script Exercise 113)

By the way…

By the way…

- …this function does the same thing:

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


(From: Script Exercise 113)

Recursive functions

- A function is called recursive if it calls itself at some point
- Idea: solve the problem by recursively solving easier problems.

- Task: compute the factorial of a number.
  - Mathematical definition

$$n! = \begin{cases} 1, & n = 0 \\ n(n-1)! & n \geq 1 \end{cases}$$

Factorial, iterative

$$n! = \begin{cases} 1, & n = 0 \\ n(n-1)! & n \geq 1 \end{cases}$$

```
int factorial (int n)
{
   int result = 1;

   for (int i = 1; i <= n; i++) {
      result *= i;
   }

   return result;
}
```

Factorial, recursive

$$n! = \begin{cases} 1, & n = 0 \\ n(n-1)! & n \geq 1 \end{cases}$$

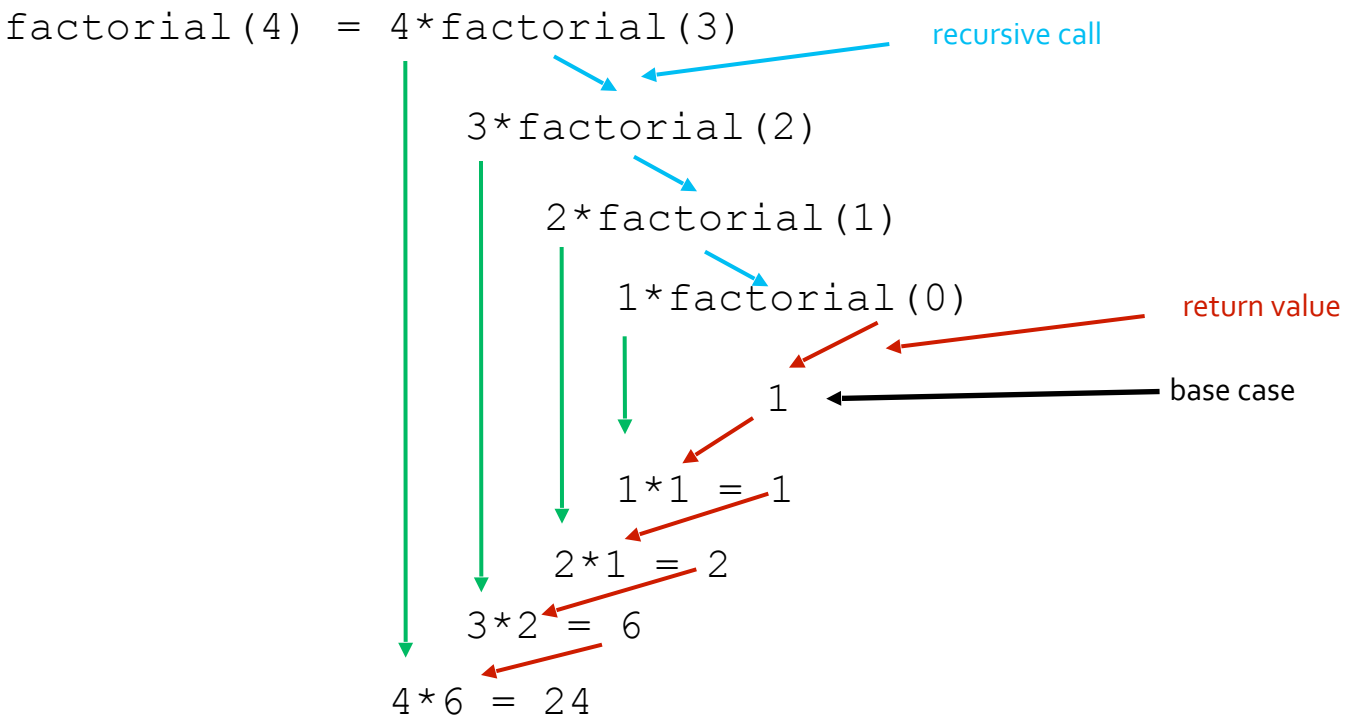
```
int factorial (int n)
{
  if (n == 0) return 1;

  return n * factorial(n-1);
}
```

base case

recursive call

Factorial, recursive execution



Power function

$$x^n = x * x^{n-1}$$

```
ifmp::integer power (const int x, const unsigned int n)
{
   if ( n == 0 ) return 0;

   if ( n == 1 ) return x;
   return x * power(x, n-1);
}
```

Recursion Exercise 1

Recursion Exercise 1

Rewrite the following recursive function in iterative form.

```
unsigned int f (const unsigned int n)
{
  if (n <= 2) return 1;
  return f(n-1) + 2 * f(n-3);
}
```


(From: Script Exercise 128)

Recursion Exercise 1

Solution below uses just 4 variables.

Other solutions are of course also possible.

```
unsigned int f_it (const unsigned int n)
{
   if (n <= 2) return 1;
   unsigned int a = 1;  // f(0)
   unsigned int b = 1;  // f(1)
   unsigned int c = 1;  // f(2)
   for (unsigned int i = 3; i < n; ++i) {
     const unsigned int a_prev = a;  // f(i-3)
     a = b;                          // f(i-2)
     b = c;                          // f(i-1)
     c = b + 2 * a_prev;             // f(i)
   }
   return c + 2 * a;                 // f(n-1) + 2 * f(n-3)
}
```


(From: Script Exercise 128)

Recursion Exercise 2

Recursion Exercise 2

Rewrite the following recursive function in iterative form.

```
unsigned int f (const unsigned int n)
{
  if (n == 0) return 1;
  return f(n-1) + 2 * f(n/2);
}
```


(From: Script Exercise 129)

Recursion Exercise 2

Solution below stores intermediate results in a vector.
Other solutions are of course also possible.

```
unsigned int f_it (const unsigned int n)
{
  if (n == 0) return 1;
  std::vector<unsigned int> f_values(n + 1, 0);
  f_values[0] = 1;
  for (unsigned int i=1; i<=n; ++i)
    f_values[i] = f_values[i-1] + 2 * f_values[i/2];
  return f_values[n];
}
```


(From: Script Exercise 129)