An Introduction to Programming

through C++

with

Manoj Prabhakaran

Lecture 23

Revision

Bugs and Loop Invariants

Using Randomness

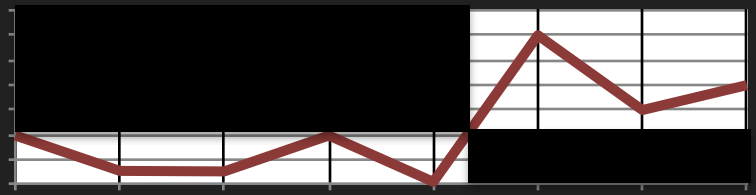
Based on material developed by Prof. Abhiram G. Ranade

Today's Example

- Given a non-empty array of numbers, rearrange it into two parts such that all numbers from the first part are ≤ x and the others are > x, where x is the first element of the array

```
Input:  99 8 7 120 300 6 99 150
Output: 99 8 7 99 6 | 300 120 150
```

- Order within each part can be arbitrary
- This problem is called "**pivoting**"
- Used in **Quicksort**



Pivot: Attempt 1

- High-level idea

```
typedef int* it;
// Input: begin points to the first element, end points beyond the last.
// Output: a pointer s.t. elements strictly to its left are the ones <= pivot
// Idea: Advance begin and end, till they both reach elements that need to be swapped
// Invariant: elements to the left of begin and right of end are finalised
it pivot(it begin, it end) {
  auto pivot = *begin; // this is the value compared against
  while(begin<end) {
    if (*begin<=pivot)  begin++;
    if (*end>pivot)  end--;
    if (*begin>pivot && *end<=pivot)  // if no progress from both the above
      std::swap(*begin,*end);         // then we make progress here
  }
  return begin;                       // the ones to the left of begin are <= pivot
}
```

Pivot: Attempt 1

- Test each function by itself: *Unit Test*

```
typedef int* it;
// Input: begin points to the first ele
// Output: a pointer s.t. elements str
// Idea: Advance begin and end, till th
// Invariant: elements to the left of b
it pivot(it begin, it end) {
  auto pivot = *begin; // this is the v
  while(begin<end) {
    if (*begin<=pivot)  begin++;
    if (*end>pivot)  end--;
    if (*begin>pivot && *end<=pivot)
      std::swap(*begin,*end);
  }
  return begin;
}
```

```
void unittest_pivot() {
    int A[8] = {99, 8, 7, 120, 300, 6, 99,  13};
    cout << "Before: ";
    for(int a: A) { cout << a << " "; }
    cout << endl;
    auto mid = pivot(A,A+8);
    cout << "After: ";
    for(it a=A; a<mid; a++) { cout << *a << " "; }
    cout << " | " ;
    for(it a=mid; a<A+8; a++) { cout << *a << " "; }
    cout << endl;
}
```

```
Before: 99 8 7 120 300 6 99 13
After: 99 8 7 13 99  | 6 300 120
```

buggy!

Debugging

```
9  #define TRACK (std::cerr << __LINE__ <<  ": *begin = " << *b
10                          << " end - begin = " << end-begin << endl)
11 typedef int* it;
12 it pivot(it begin, it end) {
13   auto pivot = *begin;
14   TRACK;
15   while(begin<end) {
16     if (*begin<=pivot)  begin++;
17     if (*end>pivot)  end--;
18     TRACK;
19     if (*begin>pivot && *end<=pivot) {
20       std::swap(*begin,*end);
21       TRACK;
22     }
23   }
24   return begin;
25 }
```

Oops, didn't see this in the test runs (will surface if it's < pivot)

```
Before: 99 8 7 120 300 6 99 13
14: *begin = 99 *end = 1828500096 end-begin = 8
18: *begin = 8 *end = 13 end-begin = 6
18: *begin = 7 *end = 13 end-begin = 5
18: *begin = 120 *end = 13 end-begin = 4
21: *begin = 13 *end = 120 end-begin = 4
18: *begin = 300 *end = 99 end-begin = 2
21: *begin = 99 *end = 300 end-begin = 2
18: *begin = 6 *end = 6 end-begin = 0
After: 99 8 7 13 99  | 6 300 120
```

Pivot: Attempt 2

```
9  #define TRACK (std::cerr << __LINE__ <<  ": *begin = " << *begin << " *end = " << *end \
10                          << " end - begin = " << end-begin << endl)
11 typedef int* it;
12 it pivot(it begin, it end) {
13   auto pivot = *begin;
14   end--;
15   TRACK;
16   while(begin<end) {
17
18
19
20                                                                    {
21
22
23
24
25
26
```

Loop invariant:

At the start of the loop, positions < begin and > end are already finalised. begin and end are pointing to positions which are not yet moved into the correct part.

What if begin==end when exiting the loop?

Loop body will need to be executed once more to advance one of begin and end.

```
Before: 99 8 7 120 300 6 99 13
15: *begin = 99 *end = 13 end-begin = 7
19: *begin = 8 *end = 13 end-begin = 6
19: *begin = 7 *end = 13 end-begin = 5
19: *begin = 120 *end = 13 end-begin = 4
22: *begin = 13 *end = 120 end-begin = 4
19: *begin = 300 *end = 99 end-begin = 2
22: *begin = 99 *end = 300 end-begin = 2
19: *begin = 6 *end = 6 end-begin = 0
After: 99 8 7 13 99  | 6 300 120
```

Still buggy

Pivot: Attempt 3

```
9  #define TRACK (std::cerr << __LINE__ <<  ": *begin = " << *begin << " *end = " << *end \
10                          << " end - begin = " << end-begin << endl)
11 typedef int* it;
12 it pivot(it begin, it end) {
13   auto pivot = *begin;
14   end--;
15   TRACK;
16   while(begin <= end) {
17     if (*begin<=pivot)  begin++;
18     if (*end>pivot)  end--;
19     TRACK;
20     if (*begin>pivot && *end<=pivot)
21       std::swap(*begin,*end);
22       TRACK;
23     }
24   }
25   return begin;
26 }
```

```
Before: 99 8 7 120 300 6 99 13
15: *begin = 8 *end = 13 end-begin = 6
19: *begin = 7 *end = 13 end-begin = 5
19: *begin = 120 *end = 13 end-begin = 4
22: *begin = 13 *end = 120 end-begin = 4
19: *begin = 300 *end = 99 end-begin = 2
22: *begin = 99 *end = 300 end-begin = 2
19: *begin = 6 *end = 6 end-begin = 0
19: *begin = 300 *end = 6 end-begin = -1
22: *begin = 6 *end = 300 end-begin = -1
After: 99 8 7 13 99 300  | 6 120
```

shouldn't swap after begin crosses end!

Still buggy!

Pivot: Fixed

```
9  #define TRACK (std::cerr << __LINE__ <<  ": *begin = " << *begin << " *end = " << *end \
10                          << " end - begin = " << end-begin << endl)
11 typedef int* it;
12 it pivot(it begin, it end) {
13   auto pivot = *begin;
14   end--;
15   TRACK;
16   while(begin <= end) {
17     if (*begin<=pivot)  begin++;
18     if (*end>pivot)  end--;
19     TRACK;
20     if (begin<end && *begin>pivot && *end<=pivot) {
21       std::swap(*begin,*end);
22       TRACK;
23     }
24   }
25   return begin;
26 }
```

Initially: By assumption on begin, end, and by decrementing end

Invariant to maintain

Elements in positions < begin, > end should be "pivoted."
If entering, begin, end should point to positions in the array.

Maintains it

Maintains it

Maintains it

Will exit the loop?

end - begin decreases in every iteration, or in the next one

If loop exited with invariant intact, fully pivoted.
Also begin points to the right of first part.

```
Before: 99 8 7 120 300 6 99 13
15: *begin = 99 *end = 13 end-begin = 7
19: *begin = 8 *end = 13 end-begin = 6
19: *begin = 7 *end = 13 end-begin = 5
19: *begin = 120 *end = 13 end-begin = 4
22: *begin = 13 *end = 120 end-begin = 4
19: *begin = 300 *end = 99 end-begin = 2
22: *begin = 99 *end = 300 end-begin = 2
19: *begin = 6 *end = 6 end-begin = 0
19: *begin = 300 *end = 6 end-begin = -1
After: 99 8 7 13 99 6  | 300 120
```

Fixed!

Review the logic using invariants

Pivot: Fixed

```
9  #define TRACK (std::cerr << __LINE__ <<  ": *begin = " << *begin << " *end = " << *end \
10                          << " end - begin = " << end-begin << endl)
11 typedef int* it;
12 it pivot(it begin, it end) {
13   auto pivot = *begin;
14   end--;
15   TRACK;
16   while(begin <= end) {
17     if (*begin<=pivot)  begin++;
18     if (*end>pivot)  end--;
19     TRACK;
20     if (begin<end && *begin>pivot && *end<=pivot) {
21       std::swap(*begin,*end);
22       TRACK;
23     }
24   }
25   return begin;
26 }
```

Can advance begin, end, more eagerly as long as the invariant will be maintained

```
Before: 99 8 7 120 300 6 99 13
15: *begin = 99 *end = 13 end-begin = 7
19: *begin = 8 *end = 13 end-begin = 6
19: *begin = 7 *end = 13 end-begin = 5
19: *begin = 120 *end = 13 end-begin = 4
22: *begin = 13 *end = 120 end-begin = 4
19: *begin = 300 *end = 99 end-begin = 2
22: *begin = 99 *end = 300 end-begin = 2
19: *begin = 6 *end = 6 end-begin = 0
19: *begin = 300 *end = 6 end-begin = -1
After: 99 8 7 13 99 6  | 300 120
```

Pivot: Final Version

```
typedef int* it;
it pivot(it begin, it end) {
  if(begin >= end)
    throw std::invalid_argument("Cannot pivot an empty range");
  auto pivot = *begin;
  begin++; end--;
  while(begin <= end) {
    if (*begin<=pivot)  begin++;
    if (*end>pivot)  end--;
    if (begin<end && *begin>pivot && *end<=pivot)
      std::swap(*begin++,*end--);
  }
  return begin;
}
```

Pivot: Final Version

```
typedef int* it;
it pivot(it begin, it end) {
  if(begin >= end)
    throw std::invalid_argument("Car
  auto pivot = *begin;
  begin++; end--;
  while(begin <= end) {
    if (*begin<=pivot)  begin++;
    if (*end>pivot)  end--;
    if (begin<end && *begin>pivot &&
      std::swap(*begin++,*end--);
  }
  return begin;
}
```

```
bool check_pivot(it begin, it end, it mid) {
  auto pivot = *begin;
  for(auto a = begin; a < mid; a++)
    if(*a > pivot) return false;
  for(auto a = mid; a < end; a++)
    if(*a <= pivot) return false;
  return true;
}
void unittest_pivot() {
  int A[8] = {99, 8, 7, 120, 300, 6, 99, 100};
  it mid = pivot(A,A+8);
  if(!check_pivot(A,A+8,mid))
    throw std::logic_error("pivot failed");
}
```

Can we check many times with random inputs?

Generating a Random Number

- From the system's "random device" (implementation dependent)

```
#include <random>
...
  std::random_device rd;          // "system's" random device
  unsigned x = rd();              // output from the device
```

- Typically slow. Possibly degrades when accessed many times quickly.
- Often used as the "seed" for a "pseudorandom number generator"

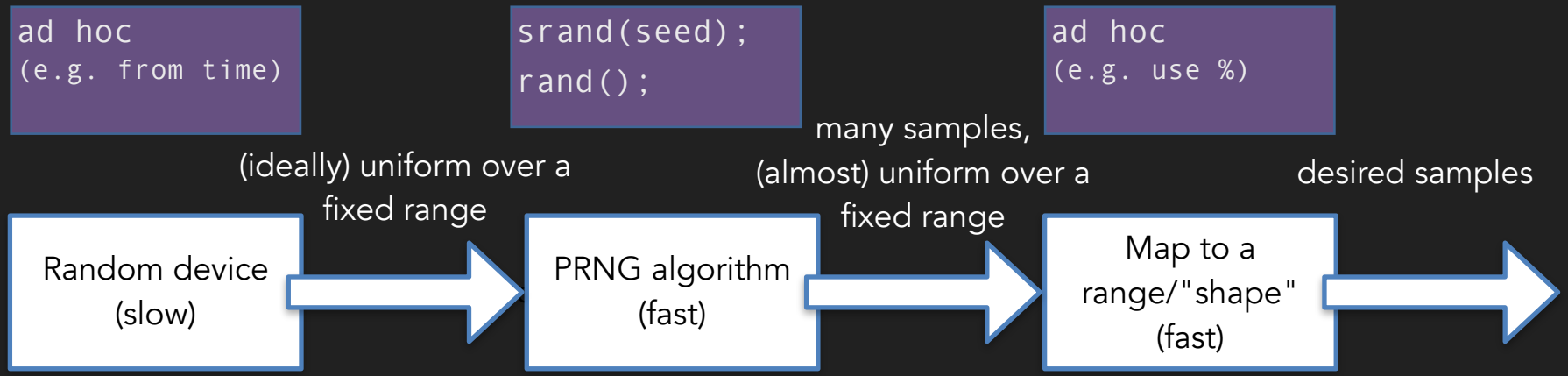
```
  srand(rd());                    // draw a sample from rd and seed a PRG
  unsigned y = rand();            // output in the range [0,RAND_MAX]
```

- Quality of `rand()` is not guaranteed by the standards. Instead can use:

```
  std::default_random_engine source(rd());          // seed a PRNG from device
  std::uniform_int_distribution<int> sampler(-7,7); // specify custom range
  int z = sampler(source);  // generate required kind of sample using given source
```

Generating a Random Number

C style



`random_device` `default_random_engine` `uniform_int_distribution<int>`

C ++ style

Only for algorithmic purposes; not meant for cryptographic purposes

others:
`minstd_rand`
`mt19937`
...

others:
`uniform_real_distribution`
`normal_distribution`
...

Generating a Random Number

- Let us wrap it in a convenient to use class

```
class dice {
  std::default_random_engine source;
  std::uniform_int_distribution<int> sampler;
public:
  // construct by passing a, b. samples will be in the range [a,b]
  dice(int a, int b)
     : source(std::random_device()()), sampler(a,b) {}
  dice(int N=2) : dice(1,N) {} // a constructor for N-sided dice
  // get the next sample using roll()
  int roll() { return sampler(source); }
  // a version of roll that ignores the original (a,b)
  int roll(int n) { return std::uniform_int_distribution<int>(1,n)(source); }
};
```

Pivot: Final Version

```
typedef int* it;
it pivot(it begin, it end) {
  if(begin >= end)
    throw std::invalid_argument("Cannot pivot an empty range");
  auto pivot = *begin;
  begin++; end--;
  while(begin <= end) {
    if (*begin<=pivot)  begin++;
    if (*end>pivot)  end--;
    if (begin<end && *begin>pivot &&
      std::swap(*begin++,*end--);
  }
  return begin;
}
```

```
void rand_unittest_pivot(int N=100, int maxval=100) {
  const int maxlen = 100;   // maximum array size
  int A[maxlen];
  dice Dlen(maxlen), Dval(maxval);
  for (int i=0; i<N; ++i) {
    int n = Dlen.roll();    // set array length
    for (int j=0; j<n; ++j)
      A[j] = Dval.roll();   // each array entry
      if(!check_pivot(A,A+n,pivot(A,A+n)))
        throw std::logic_error("pivot failed");
    }
}
```

Quicksort

- A sorting algorithm that works by pivoting repeatedly
- High-level idea:

```
void quicksort(it begin, it end) {
  if(begin >= end) return; // empty range
  // call pivot to split into left part < right part
  // recursively sort each part
}
```

Will the recursion end?

Idea: Each part would be shorter than the original

But what if the left part is all of the original array (pivot was the max element)?

Quicksort

- A sorting algorithm that works by pivoting repeatedly
- High-level idea:

```
void quicksort(it begin, it end) {
  if(begin >= end) return; // empty range
  // call pivot to split into left part < right part.
  // move pivoting element to its final position (last in the left part)
  // then consider left part to have one less element
  // now recursively sort left and right parts, in place.
}
```

Quicksort

- A sorting algorithm that works by pivoting repeatedly
- Code:

```
void quicksort(it begin, it end) {
  if(begin >= end) return; // empty range
  it mid = pivot(begin,end);
  // move pivoting element to its final position (last in the left part)
  std::swap(*begin,*(mid-1));  // *(mid-1) is the pivot value
  quicksort(begin, mid-1);     // sort {*begin, *(begin+1),...,*(mid-2)}
  quicksort(mid, end);         // sort {*mid, *(mid+1),...,*(end-1)}
}
```

Quicksort

- A sorting algorithm that works by pivoting repeatedly
- Can speed up slightly by using single element also as a base-case

```
void quicksort(it begin, it end) {
  if(begin + 1 >= end) return; // empty range or one element
  it mid = pivot(begin,end);
  // move pivoting element to its final position (last in the left part)
  std::swap(*begin,*(mid-1));  // *(mid-1) is the pivot value
  quicksort(begin, mid-1);     // sort {*begin, *(begin+1),...,*(mid-2)}
  quicksort(mid, end);         // sort {*mid, *(mid+1),...,*(end-1)}
}
```

Quicksort

- If input is already (almost) sorted, this version of quicksort makes slow progress
- Fix: Pick a random element as the pivot (instead of the first)

```
void quicksort(it begin, it end) {
  if(begin + 1 >= end) return; // empty range or one element
  it mid = pivot(begin,end);
  // move pivoting element to its final position (last in the left part)
  std::swap(*begin,*(mid-1));  // *(mid-1) is the pivot value
  quicksort(begin, mid-1);     // sort {*begin, *(begin+1),...,*(mid-2)}
  quicksort(mid, end);         // sort {*mid, *(mid+1),...,*(end-1)}
}
```

Randomised Quicksort

```
void quicksort(it begin, it end) {
  if(begin + 1 >= end) return; // empty range or one element
  int R = dice().roll(end-begin); // using the dice class we created
  std::swap(*begin,*(begin+R-1)); // move a random element to the front
  it mid = pivot(begin,end);
  // move pivoting element to its final position (last in the left part)
  std::swap(*begin,*(mid-1));  // *(mid-1) is the pivot value
  quicksort(begin, mid-1);     // sort {*begin, *(begin+1),...,*(mid-2)}
  quicksort(mid, end);         // sort {*mid, *(mid+1),...,*(end-1)}
}
```

Exercise

- Add a unit-test for sorting
- Use several randomly generated inputs
  - Add non-random effects: Inputs are already sorted or reverse sorted; repeating elements; empty array; etc.