11. Reference Types

Reference Types: Definition and Initialization, Pass By Value, Pass by Reference, Temporary Objects, Constants, Const-References

Reference Types

- We can make functions change the values of the call arguments
- no new concept for functions, but a new class of types

Reference Types: Definition

- `$T&$` has the same range of values and functionality as `$T$`, ...
- but initialization and assignment work differently.
**Reference Types: Initialization and Assignment**

- `int& darth_vader = anakin_skywalker;`  
  `darth_vader = 22; // anakin_skywalker = 22`

- A variable of reference type (a `reference`) can only be initialized with an L-Value.
- The variable is becoming an `alias` of the L-value (a different name for the referenced object).
- Assignment to the reference is to the object behind the alias.

**Reference Types: Implementation**

Internally, a value of type `T&` is represented by the address of an object of type `T`.

- `int& j; // Error: j must be an alias of something`
- `int& k = 5; // Error: the literal 5 has no address`
**Pass by Reference**

Reference types make it possible that functions modify the value of the call arguments:

```cpp
void increment (int& i)
{
    // i becomes an alias of the call argument
    ++i;
}
...
int j = 5;
increment (j);
std::cout << j << "\n"; // 6
```

---

**Pass by Value**

Formal argument does not have a reference type:

⇒ **Pass by Value**

Formal argument is initialized with the value of the actual parameter (R-Value) and thus becomes a copy.

---

**References in the Context of intervals_intersect**

// PRE: [a1, b1], [a2, b2] are (generalized) intervals,
// POST: returns true if [a1, b1], [a2, b2] intersect, in which case
//       [1, h] contains the intersection of [a1, b1], [a2, b2]
bool intervals_intersect (int& l, int& h, int a1, int b1, int a2, int b2) {
    sort (a1, b1);
    sort (a2, b2);
    l = std::max (a1, a2); // Assignments
    h = std::min (b1, b2); // via references
    return l <= h;
}
...
int lo = 0; int hi = 0;
if (intervals_intersect (lo, hi, 0, 2, 1, 3)) // Initialization
    std::cout << "[" << lo << "," << hi << "]" << "\n"; // [1,2]
References in the Context of intervals_intersect

```cpp
// POST: a <= b
void sort (int& a, int& b) {
    if (a > b)
        std::swap (a, b); // Initialization ("passing through" a, b
}

bool intervals_intersect (int& l, int& h, int a1, int b1, int a2, int b2) {
    sort (a1, b1); // Initializationsort (a2, b2); // Initializationsort (l, h); // Initializationsort (a1, b1); // Initializationsort (a2, b2); // Initializationsort (l, h);
    return l <= h;
}
```

Return by Value / Reference

- Even the return type of a function can be a reference type (return by reference)
- In this case the function call itself is an L-value

```cpp
int& increment (int& i) {
    return ++i;
}
```

Temporary Objects

What is wrong here?

```cpp
int& foo (int i) {
    return i;
}
```

```cpp
int k = 3;
int& j = foo (k); // j is an alias of a zombie
std::cout << j << "\n"; // undefined behavior
```

The Reference Guidline

Reference Guidline

When a reference is created, the object referred to must “stay alive” at least as long as the reference.
### Const-References

- have type `const T &`
- type can be interpreted as “(`const T`) &”
- can be initialized with R-Values (compiler generates a temporary object with sufficient lifetime)

```cpp
class Ivalue;
const T& r = Ivalue;
```

- `r` is initialized with the address of `Ivalue` (efficient)

```cpp
class Rvalue;
const T& r = Rvalue;
```

- `r` is initialized with the address of a temporary object with the value of the `Rvalue` (pragmatic)

### What exactly does Constant Mean?

Consider an L-value with type `const T`

- **Case 1:** `T` is no reference type
  - Then the L-value is a **constant**.

```cpp
const int n = 5;
int& i = n; // error: const-qualification is discarded
i = 6;
```

- The compiler detects our attempt to cheat

- **Case 2:** `T` is reference type.
  - Then the L-value is a read-only alias which cannot be used to change the value

```cpp
int n = 5;
const int& i = n; // i: read-only alias of n
int& j = n; // j: read-write alias
i = 6; // Error: i is a read-only alias
j = 6; // ok: n takes on value 6
```
12. Vectors I

Vector Types, Sieve of Erathostenes, Memory Layout, Iteration

Vectors: Motivation

- Now we can iterate over numbers
  
  ```cpp
  for (int i=0; i<n; ++i) ...
  ```

- Often we have to iterate over data. (Example: find a cinema in Zurich that shows “C++ Runner 2049” today)

- Vectors allow to store *homogeneous* data (example: schedules of all cinemas in Zurich)

Vectors: a first Application

The Sieve of Erathostenes

- computes all prime numbers < \( n \)
- method: cross out all non-prime numbers

The Sieve of Erathostenes

<table>
<thead>
<tr>
<th>2</th>
<th>3</th>
<th>5</th>
<th>7</th>
<th>11</th>
<th>13</th>
<th>17</th>
<th>19</th>
<th>23</th>
</tr>
</thead>
</table>

at the end of the crossing out process, only prime numbers remain.

- Question: how do we cross out numbers ??
- Answer: with a *vector*.

Sieve of Erathostenes with Vectors

```cpp
#include <iostream>
#include <vector> // standard containers with vector functionality
int main() {
    // input
    std::cout << “Compute prime numbers in \{2,...,n-1\} for n =? “;
    unsigned int n;
    std::cin >> n;
    // definition and initialization: provides us with Booleans
    // crossed_out[0],..., crossed_out[n-1], initialized to false
    std::vector<bool> crossed_out (n, false);
    // computation and output
    std::cout << “Prime numbers in \{2,...,n\}: “;
    for (unsigned int i = 2; i < n; ++i)
        if (!crossed_out[i]) { // i is prime
            std::cout << i << “ “;
            // cross out all proper multiples of i
            for (unsigned int m = 2*i; m < n; m += i)
                crossed_out[m] = true;
        }
    std::cout << “\n”;
    return 0;
}
```
Memory Layout of a Vector

- A vector occupies a contiguous memory area

Example: a vector with 4 elements

- Memory cells for a value of type $T$ each

Random Access

- The L-value $a[expr]$ has type $T$ and refers to the $i$-th element of the vector $a$ (counting from 0!)

- Random access is very efficient:
  - $p$: address of $a$
  - $p + s \cdot i$: address of $a[i]$
  - $s$: memory consumption of $T$ (in cells)

- The value $i$ of $expr$ is called index
- $[]$: subscript operator
- $a[expr]$ is an L-value
Vector Initialization

- `std::vector<int> a (5);`
  The five elements of `a` are zero initialized
- `std::vector<int> a (5, 2);`
  The 5 elements of `a` are initialized with 2.
- `std::vector<int> a {4, 3, 5, 2, 1};`
  The vector is initialized with an initialization list.
- `std::vector<int> a;`
  An initially empty vector is created.

Attention

- Accessing elements outside the valid bounds of a vector leads to undefined behavior.

```
std::vector arr (10);
for (int i=0; i<=10; ++i)
  arr[i] = 30; // runtime error: access to arr[10]!
```

Attention

- Bound Checks

  When using a subscript operator on a vector, it is the sole responsibility of the programmer to check the validity of element accesses.

```
std::vector<int> v (10);
v.at(5) = 3; // with bound check
v.push_back(8); // 8 is appended
std::vector<int> w = v; // w is initialized with v
int sz = v.size(); // sz = 11
```

Vectors are Comfortable
13. Characters and Texts I

We have seen texts before:
```cpp
std::cout << "Prime numbers in {2,...,999}:\n";
```

String-Literal

can we really work with texts? Yes:

<table>
<thead>
<tr>
<th>Character: Value of the fundamental type</th>
<th>char</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text: std::string ≈ vector of char elements</td>
<td></td>
</tr>
</tbody>
</table>

The type `char` ("character")

- represents printable characters (e.g. 'a') and control characters (e.g. '
')

```cpp
char c = 'a';
```

defines variable `c` of type `char` with value 'a' literal of type `char`

The type `char` ("character")

- is formally an integer type
- values convertible to `int / unsigned int`
- all arithmetic operators are available (with dubious use: what is 'a'/ 'b'?)
- values typically occupy 8 Bit

domain: \{-128, \ldots, 127\} or \{0, \ldots, 255\}
The ASCII-Code

- defines concrete conversion rules
  \[ \text{char} \rightarrow \text{int / unsigned int} \]
- is supported on nearly all platforms

<table>
<thead>
<tr>
<th>Zeichen</th>
<th>{0, ..., 127}</th>
</tr>
</thead>
<tbody>
<tr>
<td>'A', 'B', ..., 'Z'</td>
<td>65, 66, ..., 90</td>
</tr>
<tr>
<td>'a', 'b', ..., 'z'</td>
<td>97, 98, ..., 122</td>
</tr>
<tr>
<td>'0', '1', ..., '9'</td>
<td>48, 49, ..., 57</td>
</tr>
</tbody>
</table>

```cpp
for (char c = 'a'; c <= 'z'; ++c)
std::cout << c;
```

Einige Zeichen in UTF-8

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Codierung (jeweils 16 Bit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>🙃</td>
<td>1110111 10101111 10111001</td>
</tr>
<tr>
<td>🙄</td>
<td>11100010 10011000 10100000</td>
</tr>
<tr>
<td>🙅</td>
<td>11100010 10011000 10000011</td>
</tr>
<tr>
<td>🙆</td>
<td>11100010 10011000 10011001</td>
</tr>
<tr>
<td>A</td>
<td>01000001</td>
</tr>
</tbody>
</table>

Extension of ASCII: UTF-8

- Internationalization of Software \( \Rightarrow \) large character sets required.
  Common today: unicode, 100 symbol sets, 110000 characters.
- ASCII can be encoded with 7 bits. An eighth bit can be used to indicate the appearance of further bits.

<table>
<thead>
<tr>
<th>Bits</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0xxxxxxx</td>
</tr>
<tr>
<td>11</td>
<td>110xxxxx 10xxxxxx</td>
</tr>
<tr>
<td>16</td>
<td>1110xxxx 10xxxxxx 10xxxxxx</td>
</tr>
<tr>
<td>21</td>
<td>11110xxx 10xxxxxx 10xxxxxx 10xxxxxx</td>
</tr>
<tr>
<td>26</td>
<td>111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx</td>
</tr>
<tr>
<td>31</td>
<td>1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx</td>
</tr>
</tbody>
</table>

Interesting property: for each byte you can decide if a new UTF-8 character begins.

Einige Zeichen in UTF-8

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Codierung (jeweils 16 Bit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>🙃</td>
<td>1110111 10101111 10111001</td>
</tr>
<tr>
<td>🙄</td>
<td>11100010 10011000 10100000</td>
</tr>
<tr>
<td>🙅</td>
<td>11100010 10011000 10000011</td>
</tr>
<tr>
<td>🙆</td>
<td>11100010 10011000 10011001</td>
</tr>
<tr>
<td>A</td>
<td>01000001</td>
</tr>
</tbody>
</table>

Caesar-Code

Replace every printable character in a text by its pre-pre-predecessor.

- ' ' (32) \( \rightarrow \) '|' (124)
- '! ' (33) \( \rightarrow \) '} ' (125)
- 'D' (68) \( \rightarrow \) 'A' (65)
- 'E' (69) \( \rightarrow \) 'B' (66)
- '∼' (126) \( \rightarrow \) '} ' (123)

P.S.: Search for apple "unicode of death" P.S.: Unicode/UTF-8 is not relevant for the exam
Caesar-Code: shift-Function

// pre: divisor > 0
// post: return the remainder of dividend / divisor
// with 0 <= result < divisor
int mod(int dividend, int divisor);

// POST: if c is one of the 95 printable ASCII characters, c is
// cyclically shifted s printable characters to the right
char shift(char c, int s) {
  if (c >= 32 && c <= 126) { // c printable
    c = 32 + mod(c - 32 + s, 95);
  }
  return c;
}

"- 32" transforms interval [32, 126] to [0, 94]
"32 +" transforms interval [0, 94] back to [32, 126]
mod(x,95) is the representative of \( x \mod 95 \) in interval [0, 94]

Caesar-Code: caesar-Function

// POST: Each character read from std::cin was shifted cyclically
// by s characters and afterwards written to std::cout
void caesar(int s) {
  std::cin >> std::noskipws; // #include <ios>
  char next;
  while (std::cin >> next) {
    std::cout << shift(next, s);
  }
  return;
}

Conversion to bool: returns false if and only if the input is empty.
shifts only printable characters.

Caesar-Code: Main Program

int main() {
  int s;
  std::cin >> s;
  // Shift input by s
  caesar(s);
  return 0;
}

Encode: shift by \( n \) (here: 3)

```
Hello World, my password is 1234.
Khoor#Zruog/#p#svdzruq#lv#45671
```

Encode: shift by \(-n\) (here: -3)

```
Khoor#Zruog/#p#svdzruq#lv#45671
Hello World, my password is 1234.
```

Caesar-Code: Generalisation

void caesar(int s) {
  std::cin >> std::noskipws;
  char next;
  while (std::cin >> next) {
    std::cout << shift(next, s);
  }
}

Better: from arbitrary character source (console, file, ...) to
arbitrary character sink (console, ...)

- Currently only from std::cin to std::cout