11. Reference Types

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

Swap!

```
// POST: values of x and y have been exchanged
void swap(int& x, int& y) {
 int t = x;
 x = y;
 y = t;
}

int main() {
    int a = 2;
    int b = 1;
    swap(a, b);
    assert(a == 1 && b == 2); // ok!
}
```

Reference Types

- We can make functions change the values of the call arguments
- not a function-specific concept, but a new class of types: *reference types*

Reference Types: Definition

T& read as "*T*-reference"

underlying type

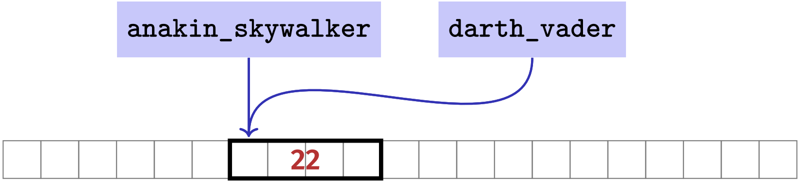
- *T*& has the same range of values and functionality as *T* ...
- ... but initialization and assignment work differently

Anakin Skywalker alias Darth Vader

Anakin Skywalker alias Darth Vader

```
int anakin_skywalker = 9;
int& darth_vader = anakin_skywalker; // Alias
darth_vader = 22;

std::cout << anakin_skywalker; // 22
```

assignment to the L-value behind the alias



Reference Types: Intialization and Assignment

```
int& darth_vader = anakin_skywalker;
darth_vader = 22; // effect: anakin_skywalker = 22
```

- A variable of reference type (a *reference*) must be initialized with an L-Value
- The variable becomes an *alias* of the L-value (a different name for the referenced object)
- Assignment to the reference updates 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*.

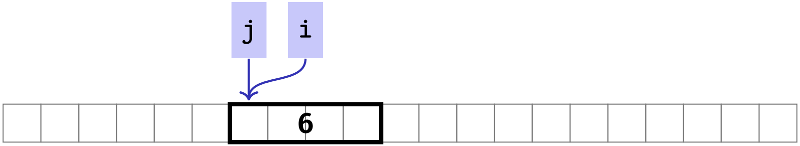
```
int& j; // Error: j must be an alias of something

int& k = 5; // Error: literal 5 has no address
```

Pass by Reference

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

```
void increment (int& i) {
  ++i;
}
...
int j = 5;
increment (j);
std::cout << j; // 6
```

initialization of the formal arguments: `i` becomes an alias of call argument `j`



Pass by Reference

Formal argument *is of* reference type:

$\Rightarrow$ *Pass by Reference*

Formal argument is (internally) initialized with the **address** of the call argument (L-value) and thus becomes an **alias**.

Pass by Value

Formal argument *is not of* 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
//       [l, 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

```
// 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); // Initialization
  sort(a2, b2); // Initialization
  l = std::max(a1, a2);
  h = std::min(b1, b2);
  return l <= h;
}
```

Return by Reference

- Even the return type of a function can be a reference type: *Return by Reference*

```
int& inc(int& i) {
  return ++i;
}
```

- call **`inc(x)`**, for some **`int`** variable **`x`**, has exactly the semantics of the pre-increment **`++x`**
- Function call *itself* now is an L-value
- Thus possible: **`inc(inc(x))`** or **`++(inc(x))`**

Temporary Objects

What is wrong here?

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

Return value of type **`int&`** becomes an alias of the formal argument (local variable **`i`**), whose memory lifetime ends after the call

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

The Reference Guidline

Reference Guideline

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)

```
const T& r = lvalue;
```

`r` is initialized with the address of *lvalue* (efficient)

```
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 L-value of type **`const`** *T*. **Case: 1** *T is no* reference type.

⇒ Then the *L-value is a constant*

```
const int n = 5;
int& a = n; // Compiler error: const-qualification discarded
a = 6;
```

The compiler detects our *cheating attempt*

What exactly does Constant Mean?

Consider L-value of type **`const`** *T*. **Case 2:** *T is* reference type.

⇒ Then the *L-value is a read-only alias* which cannot be used to change the *underlying* L-value.

```
int n = 5;

const int& r = n; // r is read-only alias of n
r = 6;            // Compiler error: read-only reference

int& rw = n;      // rw is read-write alias
rw = 6;           // OK
```

When to use `const T&`?

```
void f_1(T& arg);
```

```
void f_2(const T& arg);
```

- Argument types are references; call arguments are thus not copied, which is efficient
- But only **`f_2`** "promises" to not modify the argument

Rule

If possible, declare function argument types as **`const T&`** (*pass by read-only reference*) : efficient *and* safe.

Typically doesn't pay off for fundamental types (**`int`**, **`double`**, …). Types with a larger memory footprint will be introduced later in this course.

12. Vectors I

Vector Types, Sieve of Erathostenes, Memory Layout, Iteration

Vectors: Motivation

- Now we can iterate over numbers

```
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

2	**3**	~~4~~	**5**	~~6~~	**7**	~~8~~	~~9~~	~~10~~	**11**	~~12~~	**13**	~~14~~	~~15~~	~~16~~	**17**	~~18~~	**19**	~~20~~	~~21~~	~~22~~	**23**

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

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

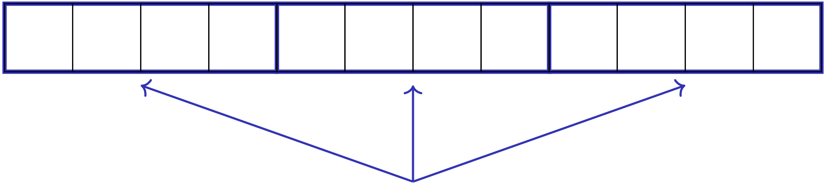
Sieve of Erathostenes with Vectors

```
#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-1 << "}:\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 3 elements of type **`T`**



Memory segments for a value of type **`T`** each

(**`T`** occupies e.g. 4 bytes)

Random Access

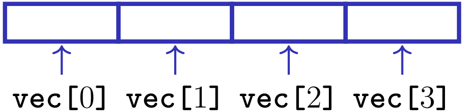
Given

- vector **`vec`** with **`T`** elements
- **`int`** expression **`exp`** with value $i \geq 0$

Then the expression

`vec [ exp ]`

- is an *L-value* of type **`T`**
- that refers to the ith element **`vec`** (counting from 0!)



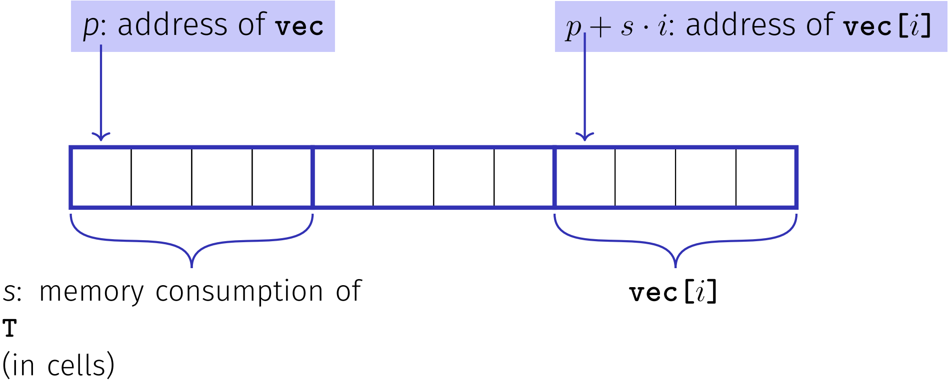
Random Access

```
vec [ exp ]
```

- The value i of **`exp`** is called *index*
- **`[]`** is the *index operator* (also *subscript operator*)

Random Access

Random access is very efficient:



Vector Initialization

- **`std::vector<int> vec(5);`**
  The five elements of **`vec`** are intialized with zeros)
- **`std::vector<int> vec(5, 2);`**
  the 5 elements of **`vec`** are initialized with 2
- **`std::vector<int> vec{4, 3, 5, 2, 1};`**
  the vector is initialized with an *initialization list*
- **`std::vector<int> vec;`**
  An initially empty vector is initialized

Attention

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

```
std::vector vec(10);
for (unsigned int i = 0; i <= 10; ++i)
  vec[i] = 30; // Runtime error: accessing vec[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.

Vectors Offer Great Functionality

Here a few example functions, additional follow later in the course.

```
std::vector<int> v(10);
std::cout << v.at(10);
  // Access with index check → runtime error
  // Ideal for homework
```

```
v.push_back(-1); // -1 is appended (added at end)
std::cout << v.size(); // outputs 11
std::cout << v.at(10); // outputs -1
```

13. Characters and Texts I

Characters and Texts, ASCII, UTF-8, Caesar Code

Characters and Texts

- We have seen texts before:

```
std::cout << "Prime numbers in {2,...,999}:\n";
```

String-Literal

- can we really work with texts? Yes!

Character:	Value of the fundamental type **`char`**
Text:	**`std::string`** ≈ vector of **`char`** elements

The type `char` ("character")

Represents printable characters (e.g. `'a'`) and *control characters* (e.g. `'\n'`)

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

Declares and initialises 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 `char` $\longrightarrow$ `(unsigned) int`

  Zeichen $\longrightarrow \{0, \ldots, 127\}$

  `'A', 'B', ... , 'Z'` $\longrightarrow 65, 66, ..., 90$
  `'a', 'b', ... , 'z'` $\longrightarrow 97, 98, ..., 122$
  `'0', '1', ... , '9'` $\longrightarrow 48, 49, ..., 57$

- Is supported on all common computer systems
- Enables arithmetic over characters

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

Extension of ASCII: Unicode, UTF-8

- Internationalization of Software $\Rightarrow$ large character sets required. Thus common today:
  - Character set *Unicode*: 150 scripts, ca. 137,000 characters
  - encoding standard *UTF-8*: mapping characters $\leftrightarrow$ numbers
- UTF-8 is a *variable-width encoding*: Commonly used characters (e.g. Latin alphabet) require only one byte, other characters up to four
- Length of a character's byte sequence is encoded via bit patterns

Useable Bits	Bit patterns
7	`0xxxxxxx`
11	`110xxxxx 10xxxxxx`
16	`1110xxxx 10xxxxxx 10xxxxxx`
21	`11110xxx 10xxxxxx 10xxxxxx 10xxxxxx`

Some Unicode characters in UTF-8

Symbol	Codierung (jeweils 16 Bit)
ﯹ	11101111 10101111 10111001
☠	11100010 10011000 10100000
☃	11100010 10011000 10000011
☙	11100010 10011000 10011001
A	01000001

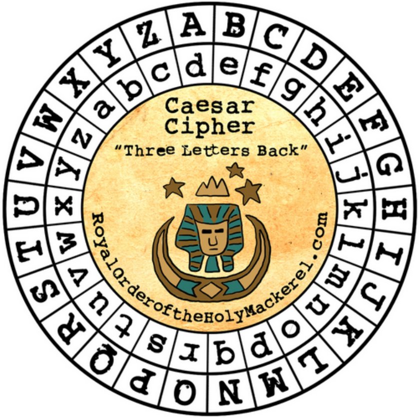
P.S.: Search for **`apple "unicode of death"`** P.S.: Unicode & UTF-8 are not relevant for the exam

http://t-a-w.blogspot.ch/2008/12/funny-characters-in-unicode.html

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)
		...		
$\sim$	(126)	$\rightarrow$	'{'	(123)



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 is printable
      c = 32 + mod(c - 32 + s,95);
    }

    return c;
}
```

"**- 32**" transforms interval $[32,126]$ to $[0,94]$
"**mod(**x**, 95)**" computes $x \bmod 95$ in $[0,94]$
"**32 +**" transforms $[0,94]$ back to $[32,126]$

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);
  }
}
```

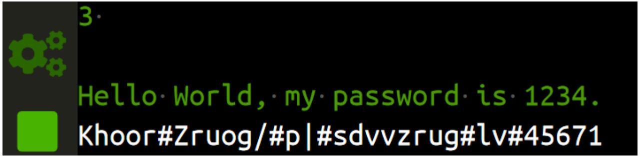
Conversion to **bool**: returns *false* if and only if the input is empty

Shift printable characters by **s**

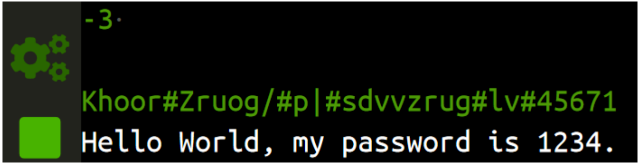
Caesar-Code: Main Program

```cpp
int main() {
  int s;
  std::cin >> s;

  // Shift input by s
  caesar(s);

  return 0;
}
```

Encode: shift by n (here: 3)



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



Caesar-Code: Generalisation

```
void caesar(int s) {
  std::cin >> std::noskipws;

  char next;
  while (std::cin >> next) {
    std::cout << shift(next, s);
  }
}
```

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

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