11. Reference Types

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

Swap!

```
// POST: values of x and y are 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
- no new concept for functions, 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.

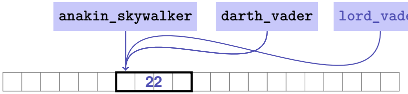
Anakin Skywalker alias Darth Vader

Anakin Skywalker alias Darth Vader

```
int anakin_skywalker = 9;
int& darth_vader = anakin_skywalker; //  alias
int& lord_vader = darth_vader; // another alias
darth_vader = 22;
```

assignment to the L-value behind the alias

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



Reference Types: Intialization 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*.

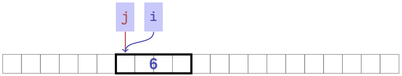
```
int& j;  // Error: j must be an alias of something

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

Call by Reference

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

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



Call by Reference

Formal argument has reference type:

⇒ **Call by Reference**

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

Call by Value

Formal argument does not have a reference type:

⇒ **Call 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 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

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

exactly the semantics of the pre-increment

Temporary Objects

What is wrong here?

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

Return value of type `int&` becomes an alias of the formal argument. But the memory lifetime of i ends after the call!

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

The Compiler as Your Friend: Constants

Constants

- are variables with immutable value

```
const int speed_of_light = 299792458;
```

- Usage: `const` before the definition

The Compiler as Your Friend: Constants

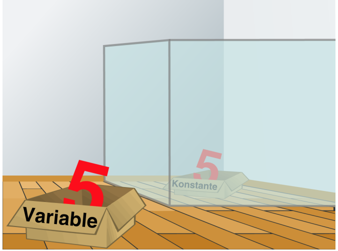
- Compiler checks that the `const`-promise is kept

```
const int speed_of_light = 299792458;
...
speed_of_light = 300000000;
```

compiler: error

- Tool to avoid errors: constants guarantee the promise :*“value does not change”*

Constants: Variables behind Glass



The `const`-guideline

`const`-guideline

For *each variable*, think about whether it will change its value in the lifetime of a program. If not, use the keyword `const` in order to make the variable a constant.

A program that adheres to this guideline is called `const`-correct.

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 an L-value with type `const` *T*

- Case 1: *T* is no reference type

Then the L-value is a constant.

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

The compiler detects our attempt to cheat

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 value

```
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
```


When `const` *T*`&` ?

Rule

Argument type `const` *T* `&` (call by *read-only* reference) is used for efficiency reasons instead of *T* (call by value), if the type *T* requires large memory. For fundamental types (`int`, `double`,...) it does not pay off.

Examples will follow later in the course

12. Arrays I

Array Types, Sieve of Erathostenes, Memory Layout, Iteration, Vectors, Characters and Texts, ASCII, UTF-8, Caesar-Code

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

Arrays: 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 an *array*.

Sieve of Erathostenes: Initialization

```
const unsigned int n = 1000;
bool crossed_out[n];
for (unsigned int i = 0; i < n; ++i)
    crossed_out[i] = false;
```

constant!

`crossed_out[i]` indicates if i has been crossed out.

Sieve of Eratosthenes: Computation

```
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;
    }
}
```

The sieve: go to the next non-crossed out number i (this must be a prime number), output the number and cross out all proper multiples of i

Arrays: Definition

Declaration of an array variable:

T a [expr]

base type

variable of array type

constant integer expression
value provides the length of the array

value is known at compile time.
e.g. literal, constant

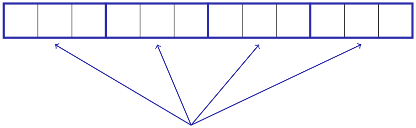
type of `a`*: „T[k]"*
values range of `a`*:* T^k

Beispiel: `bool crossed_out[n]`

Memory Layout of an Array

- An array occupies a *contiguous* memory area

example: an array with 4 elements



memory cells for a value of type `T` each

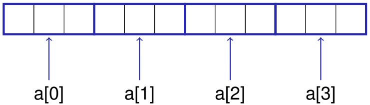
Random Access

The L-value

value i

a [expr]

has type *T* and refers to the i-th element of the array *a* (counting from 0!)



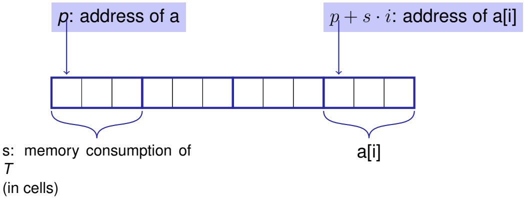
Random Access

a [*expr*]

The value i of *expr* is called *array index*.
`[]`: subscript operator

Random Access

- Random access is very efficient:



Array Initialization

- `int a[5];`

  The five elements of `a` remain uninitialized (values can be assigned later)

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

  the 5 elements of `a` are initialized with an *initialization list*.

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

  also ok: the compiler will deduce the length

Arrays are Primitive

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

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

Arrays are Primitive

Array Bound Checks

With no special compiler or runtime support it is the sole *responsibility of the programmer* to check the validity of element accesses.

Arrays are Primitive (II)

- Arrays cannot be initialized and assigned to like other types

```
int a[5] = {4,3,5,2,1};
int b[5];
b = a;          // Compiler error!
int c[5] = a;   // Compiler error!
```

Why?

Arrays are Primitive

- Arrays are legacy from the language C and primitive from a modern viewpoint
- Arrays are very low level and thus efficient
- Missing array bound checks have far reaching consequences: code with non-permitted but possible index accesses has been exploited (far too) often for malware
- the standard library offers comfortable alternatives

Vectors

- Obvious disadvantage of static arrays: *constant array length*

```
const unsigned int n = 1000;
bool crossed_out[n];
```

- remedy: use the type `Vector` from the standard library

```
#include <vector>
...
std::vector<bool> crossed_out (n, false);
```

Initialization with n elements initial value `false`.

element type in triangular brackets

Sieve of Erathostenes with Vectors

```
#include <iostream>
#include <vector> // standard containers with array 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;
```


Characters and Texts

- We have seen texts before:

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

  (`"Prime numbers in {2,...,999}:\n"`: String-Literal)

- can we really work with texts? Yes:

Character:	Value of the fundamental type **`char`**
Text:	Array with base type **`char`**

The type `char` ("character")

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

```
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
  `char` ⟶ `int` / `unsigned int`
- is supported on nearly all platforms

> Zeichen $\longrightarrow \{0, \ldots, 127\}$
> `'A', 'B', ... , 'Z'` $\longrightarrow 65, 66, ..., 90$
> `'a', 'b', ... , 'z'` $\longrightarrow 97, 98, ..., 122$
> `'0', '1', ... , '9'` $\longrightarrow 48, 49, ..., 57$

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


Extension of ASCII: UTF-8

- Internationalization of Software ⇒ 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.

Bits	Encoding
7	`0xxxxxxx`
11	`110xxxxx 10xxxxxx`
16	`1110xxxx 10xxxxxx 10xxxxxx`
21	`11110xxx 10xxxxxx 10xxxxxx 10xxxxxx`
26	`111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx`
31	`1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx`

Interesting property: for each byte you can decide if a new UTF8 character begins.

Einige Zeichen in UTF-8

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

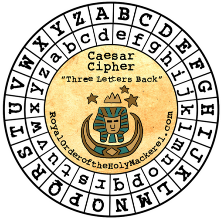
P.S.: Search for `apple "unicode of death"`

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)	→	'|'	(124)
'!'	(33)	→	'}'	(125)
		...		
'D'	(68)	→	'A'	(65)
'E'	(69)	→	'B'	(66)
		...		
∼	(126)	→	'{'	(123)



Caesar-Code: Main Program

```
// Program: caesar_encrypt.cpp
// encrypts a text by applying a cyclic shift of −3

#include<iostream>
#include<cassert>
#include<ios> // for std::noskipws

// PRE: −95 < s < 95
// POST: if c is one of the 95 printable ASCII characters, c is
//       cyclically shifted s printable characters to the right
void shift (char& c, int s);
```

spaces and newline characters shall *not* be ignored

Caesar-Code: Main Program

```
int main ()
{
  std::cin >> std::noskipws; // don’t skip whitespaces!

  // encryption loop
  char next;
  while (std::cin >> next) {
    shift (next, −3);
    std::cout << next;
  }
  return 0;
}
```

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

shifts only printable characters.

Caesar-Code: shift-Function

```
// PRE: −95 < s < 95
// POST: if c is one of the 95 printable ASCII characters, c is
//       cyclically shifted s printable characters to the right
void shift (char& c, int s)
{
  assert (−95 < s && s < 95);
  if (c >= 32 && c <= 126) {
    if (c + s > 126)
      c += (s − 95);
    else if (c + s < 32)
      c += (s + 95);
    else
      c += s;
  }
}
```

Call by reference!

Overflow – 95 backwards!

underflow – 95 forward!

normal shift

./caesar_encrypt < power8.cpp

```
„|Moldo^j7|mltbo5+‘mm
„|O^fpb|^|krj_bo|ql|qeb|bfdeqe|mltbo+

 fk‘irab|9flpqob^j;|

fkq|j^fk%&
x
||„|fkmrq
||pqa77‘lrq|99|~@ljmrqb|^[5|clo|^|:<|~8||
||fkq|^8
||pqa77‘fk|;;|^8

||„|‘ljmrq^qflk
||fkq|_|:|^|’|^8|„|_|:|^[/
||_|:|_|’|_8|||||„|_|:|^[1

||„|lrqmrq|_|’|_)|f+b+)|^[5
||pqa77‘lrq|99|^|99|~[5|:|~|99|_|’|_|99|~+Yk~8
||obqrok|-8
z
```

Program = Moldo^j

Caesar-Code: Decryption

```
// decryption loop
char next;
while (std::cin >> next) {
  shift (next, 3);
  std::cout << next;
}
```

Now: shift by 3 to *right*

An interesting way to output `power8.cpp`

- `./caesar_encrypt < power8.cpp | ./caeser_decrypt`