

9. Referenztypen

Referenztypen: Definition und Initialisierung, Call By Value , Call by Reference, Temporäre Objekte, Konstanten, Const-Referenzen

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! 😊
}
```

255

256

Referenztypen

- Wir können Funktionen in die Lage versetzen, die Werte ihrer Aufrufargumente zu ändern!
- Kein neues Konzept auf der Funktionenseite, sondern eine neue Klasse von Typen

Referenztypen

Referenztypen: Definition

$T\&$

Gelesen als „ T -Referenz“

Zugrundeliegender Typ

- $T\&$ hat den gleichen Wertebereich und gleiche Funktionalität wie T , ...
- nur Initialisierung und Zuweisung funktionieren anders.

257

258

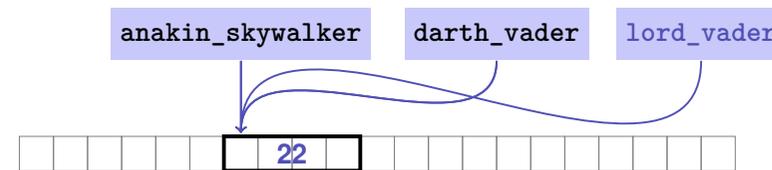
Anakin Skywalker alias Darth Vader



Anakin Skywalker alias Darth Vader

```
int anakin_skywalker = 9;
int& darth_vader = anakin_skywalker; // Alias
int& lord_vader = darth_vader; // noch ein Alias
darth_vader = 22;
std::cout << anakin_skywalker; // 22
```

Zuweisung an den L-Wert hinter dem Alias



259

260

Referenztypen: Initialisierung & Zuweisung

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

- Eine Variable mit **Referenztyp** (eine *Referenz*) kann nur mit einem **L-Wert** initialisiert werden.
- Die Variable wird dabei ein *Alias* des **L-Werts** (ein anderer Name für das referenzierte Objekt).
- Zuweisung an die Referenz erfolgt an das **Objekt** hinter dem Alias.

261

Referenztypen: Realisierung

Intern wird ein Wert vom Typ $T&$ durch die Adresse eines Objekts vom Typ T repräsentiert.

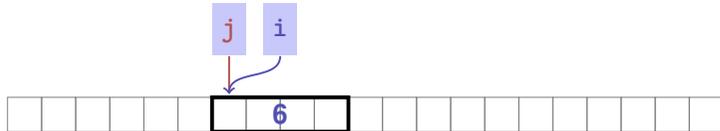
```
int& j; // Fehler: j muss Alias von irgendetwas sein
int& k = 5; // Fehler: Das Literal 5 hat keine Adresse
```

262

Call by Reference

Referenztypen erlauben Funktionen, die Werte ihrer Aufrufargumente zu ändern:

```
void increment (int& i) ← Initialisierung der formalen Argumente
{ // i wird Alias des Aufrufarguments
  ++i;
}
...
int j = 5;
increment (j);
std::cout << j << "\n"; // 6
```



263

Call by Reference

Formales Argument hat Referenztyp:

⇒ **Call by Reference**

Formales Argument wird (intern) mit der *Adresse* des Aufrufarguments (L-Wert) initialisiert und wird damit zu einem *Alias*.

264

Call by Value

Formales Argument hat keinen Referenztyp:

⇒ **Call by Value**

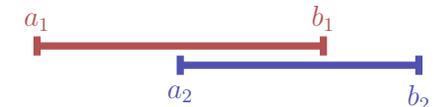
Formales Argument wird mit dem *Wert* des Aufrufarguments (R-Wert) initialisiert und wird damit zu einer *Kopie*.

265

Im Kontext: Zuweisung an Referenzen

```
// 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);
    h = std::min (b1, b2);
    return l <= h;
}
```



```
...
int lo = 0; int hi = 0;
if (intervals_intersect (lo, hi, 0, 2, 1, 3))
    std::cout << "[" << lo << ", " << hi << "]" << "\n"; // [1,2]
```

266

Im Kontext: Initialisierung von Referenzen

```
// POST: a <= b
void sort (int& a, int& b) {
    if (a > b)
        std::swap (a, b); // 'Durchreichen' der Referenzen a, b
}

bool intervals_intersect (int& l, int& h,
                          int a1, int b1, int a2, int b2) {
    sort (a1, b1); // Erzeugung von Referenzen auf a1, b1
    sort (a2, b2); // Erzeugung von Referenzen auf a2, b2
    l = std::max (a1, a2);
    h = std::min (b1, b2);
    return l <= h;
}
```

267

Return by Value / Reference

- Auch der Rückgabetyt einer Funktion kann ein Referenztyp sein (return by reference)
- In diesem Fall ist der Funktionsaufruf selbst ein L-Wert

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

Exakt die Semantik des Prä-Inkrement

268

Temporäre Objekte

Was ist hier falsch?

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

Rückgabewert vom Typ `int&` wird Alias des formalen Arguments, dessen Speicherdauer aber nach Auswertung des Funktionsaufrufes endet.

```
int k = 3;
int& j = foo (k); // j ist Alias einer "Leiche"
std::cout << j << "\n"; // undefined behavior
```

269

Die Referenz-Richtlinie

Referenz-Richtlinie

Wenn man eine Referenz erzeugt, muss das Objekt, auf das sie verweist, mindestens so lange „leben“ wie die Referenz selbst.

270

Der Compiler als Freund: Konstanten

Konstanten

- sind Variablen mit unveränderbarem Wert

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

- Verwendung: `const` vor der Definition

Der Compiler als Freund: Konstanten

- Compiler kontrolliert Einhaltung des `const`-Versprechens

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

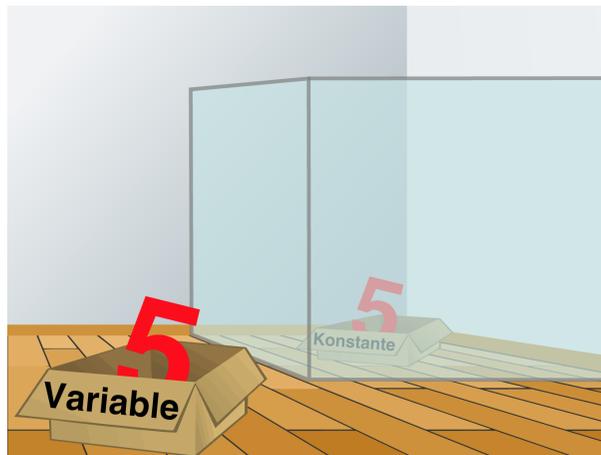
Compilerfehler!

- Hilfsmittel zur Vermeidung von Fehlern: Konstanten erlauben garantierte Einhaltung des Versprechens *“Wert ändert sich nicht”*

271

272

Konstanten: Variablen hinter Glas



Die `const`-Richtlinie

`const`-Richtlinie

Denke bei *jeder Variablen* darüber nach, ob sie im Verlauf des Programmes jemals ihren Wert ändern wird oder nicht! Im letzteren Falle verwende das Schlüsselwort `const`, um die Variable zu einer Konstanten zu machen!

Ein Programm, welches diese Richtlinie befolgt, heisst `const`-korrekt.

273

274

Const-Referenzen

- haben Typ `const T &` (= `const (T &)`)
- können auch mit R-Werten initialisiert werden (Compiler erzeugt temporäres Objekt ausreichender Lebensdauer)

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

r wird mit der Adresse von *lvalue* initialisiert (effizient)

```
const T& r = rvalue;
```

r wird mit der Adresse eines temporären Objektes vom Wert des *rvalue* initialisiert (flexibel)

275

Was genau ist konstant?

Betrachte L-Wert vom Typ `const T`

- Fall 1: *T* ist kein Referenztyp

Dann ist der L-Wert eine **Konstante**.

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

Der Schummelversuch wird vom Compiler erkannt

276

Was genau ist konstant?

Betrachte L-Wert vom Typ `const T`

- Fall 2: *T* ist Referenztyp

Dann ist der L-Wert ein Lese-Alias, **durch den der Wert dahinter nicht verändert werden darf**.

```
int n = 5;  
const int& i = n; // i: Lese-Alias von n  
int& j = n;      // j: Lese-Schreib-Alias  
i = 6;         // Fehler: i ist Lese-Alias  
j = 6;         // ok: n bekommt Wert 6
```

277

Wann `const T&` ?

Regel

Argumenttyp `const T &` (call by *read-only* reference) wird aus Effizienzgründen anstatt *T* (call by value) benutzt, wenn der Typ *T* grossen Speicherbedarf hat. Für fundamentale Typen (`int`, `double`,...) lohnt es sich aber nicht.

Beispiele folgen später in der Vorlesung

278

10. Felder (Arrays) I

Feldtypen, Sieb des Eratosthenes, Speicherlayout, Iteration, Vektoren, Zeichen und Texte, ASCII, UTF-8, Caesar-Code

Felder: Motivation

- Wir können jetzt über Zahlen iterieren

```
for (int i=0; i<n ; ++i) ...
```

- Oft muss man aber über *Daten* iterieren (Beispiel: Finde ein Kino in Zürich, das heute „C++ Runner 2049“ zeigt)
- Felder dienen zum Speichern *gleichartiger* Daten (Beispiel: Spielpläne aller Zürcher Kinos)

279

280

Felder: erste Anwendung

Das Sieb des Eratosthenes

- berechnet alle Primzahlen $< n$
- Methode: Ausstreichen der Nicht-Primzahlen

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

Am Ende des Streichungsprozesses bleiben nur die Primzahlen übrig.

- Frage: wie streichen wir Zahlen aus ??
- Antwort: mit einem *Feld* (Array).

Sieb des Eratosthenes: Initialisierung

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

Konstante!

`crossed_out[i]` gibt an, ob `i` schon ausgestrichen wurde.

281

282

Sieb des Eratosthenes: Berechnung

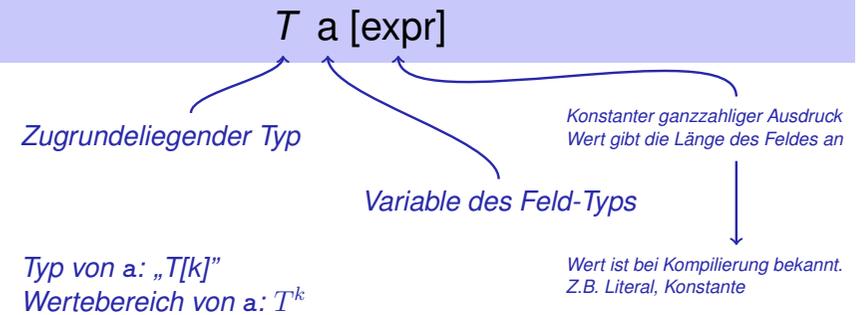
```
for (unsigned int i = 2; i < n; ++i)
    if (!crossed_out[i] ){
        // i is prime
        std::cout << i << " ";
        // cross out all proper multiples of i
        for (unsigned int m = 2*i; m < n; m += i)
            crossed_out[m] = true;
    }
}
```

Das Sieb: gehe zur jeweils nächsten nichtgestrichenen Zahl i (diese ist Primzahl), gib sie aus und streiche alle echten Vielfachen von i aus.

283

Felder: Definition

Deklaration einer Feldvariablen (array):



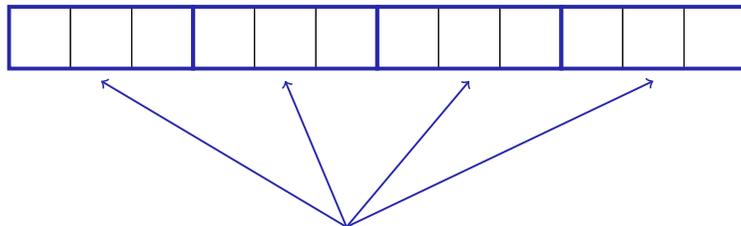
Beispiel: `bool crossed_out[n]`

284

Speicherlayout eines Feldes

- Ein Feld belegt einen *zusammenhängenden* Speicherbereich

Beispiel: ein Feld mit 4 Elementen

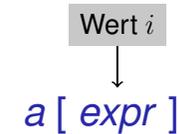


Speicherzellen für jeweils einen Wert vom Typ T

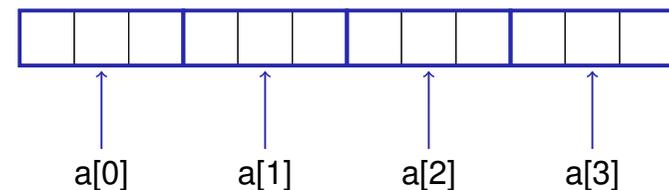
285

Wahlfreier Zugriff (Random Access)

Der L-Wert



hat Typ T und bezieht sich auf das i -te Element des Feldes a (Zählung ab 0!)



286

Wahlfreier Zugriff (Random Access)

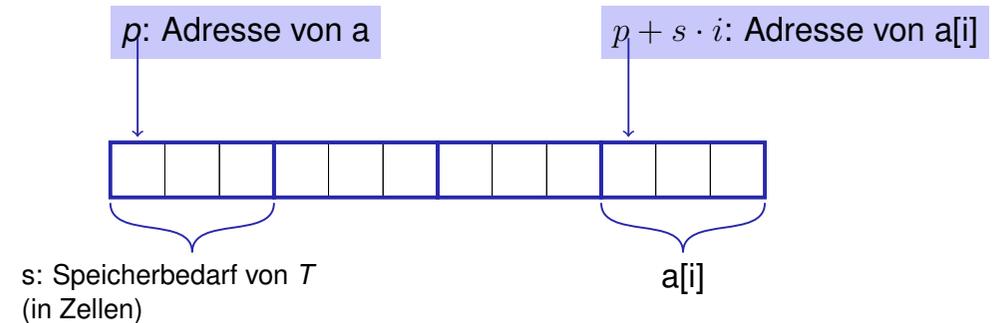
$a[expr]$

Der Wert i von $expr$ heisst *Feldindex*.

$[\]$: Subskript-Operator

Wahlfreier Zugriff (Random Access)

- Wahlfreier Zugriff ist sehr effizient:



287

288

Feld-Initialisierung

- `int a[5];`

Die 5 Elemente von a bleiben uninitialisiert (können später Werte zugewiesen bekommen)

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

Die 5 Elemente von a werden mit einer *Initialisierungsliste* initialisiert.

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

Auch ok: Länge wird vom Compiler deduziert

Felder sind primitiv

- Der Zugriff auf Elemente ausserhalb der gültigen Grenzen eines Feldes führt zu undefiniertem Verhalten.

```
int arr[10];
for (int i=0; i<=10; ++i)
    arr[i] = 30; // Laufzeit-Fehler: Zugriff auf arr[10]!
```

289

290

Felder sind primitiv

Prüfung der Feldgrenzen

In Abwesenheit spezieller Compiler- oder Laufzeitunterstützung ist es die alleinige *Verantwortung des Programmierers*, die Gültigkeit aller Elementzugriffe zu prüfen.

Felder sind primitiv (II)

- Man kann Felder nicht wie bei anderen Typen initialisieren und zuweisen:

```
int a[5] = {4,3,5,2,1};
int b[5];
b = a;           // Fehlermeldung des Compilers!
int c[5] = a;    // Fehlermeldung des Compilers!
```

Warum?

291

292

Felder sind primitiv

- Felder sind „Erblast“ der Sprache C und aus heutiger Sicht primitiv.
- In C sind Felder sehr maschinennah und effizient, bieten aber keinen „Luxus“ wie eingebautes Initialisieren und Kopieren.
- Fehlendes Prüfen der Feldgrenzen hat weitreichende Konsequenzen. Code mit nicht erlaubten aber möglichen Index-Zugriffen wurde von Schadsoftware schon (viel zu) oft ausgenutzt.
- Die Standard-Bibliothek bietet komfortable Alternativen

Vektoren

- Offensichtlicher Nachteil statischer Felder: *konstante Feldlänge*

```
const unsigned int n = 1000;
bool crossed_out[n];
```
- Abhilfe: Verwendung des Typs `vector` aus der Standardbibliothek

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

Initialisierung mit n Elementen
Initialwert `false`.

↑
Elementtyp, in spitzen Klammern

293

294

Sieb des Eratosthenes mit Vektoren

```
#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;
}
```

297

Zeichen und Texte

- Texte haben wir schon gesehen:

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

String-Literal

- Können wir auch „richtig“ mit Texten arbeiten? Ja:

Zeichen: Wert des fundamentalen Typs `char`
Text: Feld mit zugrundeliegendem Typ `char`

298

Der Typ `char` (“character”)

- repräsentiert druckbare Zeichen (z.B. `'a'`) und *Steuerzeichen* (z.B. `'\n'`)

`char c = 'a'`

↑
definiert Variable c vom Typ char mit Wert 'a'

↑
Literal vom Typ char

299

Der Typ `char` (“character”)

ist formal ein ganzzahliger Typ

- Werte konvertierbar nach `int` / `unsigned int`
- Alle arithmetischen Operatoren verfügbar (Nutzen zweifelhaft: was ist `'a'` / `'b'` ?)
- Werte belegen meistens 8 Bit

Wertebereich:
{-128, ..., 127} oder {0, ..., 255}

300

Der ASCII-Code

- definiert konkrete Konversionsregeln
char → int / unsigned int
- wird von fast allen Plattformen benutzt

Zeichen → {0, ..., 127}

'A', 'B', ... , 'Z' → 65, 66, ..., 90

'a', 'b', ... , 'z' → 97, 98, ..., 122

'0', '1', ... , '9' → 48, 49, ..., 57

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

301

Erweiterung von ASCII: UTF-8

- Internationalisierung von Software ⇒ grosse Zeichensätze nötig.
Heute üblich: Unicode, 100 Schriftsysteme, 110000 Zeichen.
- ASCII kann mit 7 Bits codiert werden. Ein achttes Bit ist verwendbar, um das Vorkommen weiterer Bits festzulegen.

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

Interessante Eigenschaft: bei jedem Byte kann entschieden werden, ob ein UTF8 Zeichen beginnt.

302

Einige Zeichen in UTF-8

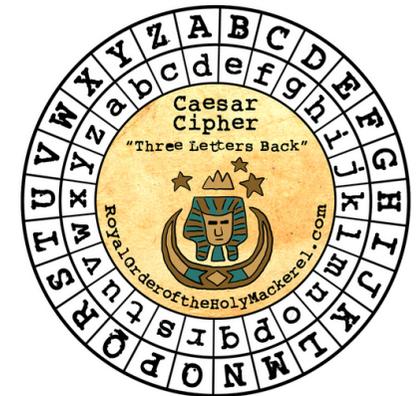
Symbol	Codierung (jeweils 16 Bit)
☠	11100010 10011000 10100000
☺	11100010 10011000 10000011
⋮	11100010 10001101 10101000
☹	11100010 10011000 10011001
🇧🇩	11100011 10000000 10100000
ع	11101111 10101111 10111001

©http://a-w.blogspot.ch/2008/12/lunny-characters-in-unicode.html

Caesar-Code

Ersetze jedes druckbare Zeichen in einem Text durch seinen Vor-Vor-Vorgänger.

- ' ' (32) → '|' (124)
- '|' (33) → '}' (125)
- ...
- 'D' (68) → 'A' (65)
- 'E' (69) → 'B' (66)
- ...
- ~ (126) → '{' (123)



304

Caesar-Code:

Hauptprogramm

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

#include<iostream>
#include<cassert>
#include<ios> // for std::noskipws ← Leerzeichen und Zeilen-
umbrüche sollen nicht ignoriert werden

// PRE: s < 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);
```

305

Caesar-Code:

Hauptprogramm

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

    // encryption loop
    char next;
    while (std::cin >> next) ← Konversion nach bool:
    {                                     liefert false genau dann,
        shift (next, -3);                wenn die Eingabe leer ist.
    }
    return 0;
}
// Verschiebt nur druck-
// bare Zeichen.
```

306

Caesar-Code:

shift-Funktion

```
// PRE: s < 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) ← Call by reference!
{
    assert (s < 95 && s > -95);
    if (c >= 32 && c <= 126) {
        if (c + s > 126) ← Überlauf - 95 zurück!
            c += (s - 95);
        else if (c + s < 32) ← Unterlauf - 95 vorwärts!
            c += (s + 95);
        else ← Normale Verschiebung
            c += s;
    }
}
```

307

./caesar_encrypt < power8.cpp

```
„|Moldo^j7+mltbo5+'mm
„|0^fpbl^|krj_bo|ql|qeb|bfdeqe|mltbo+
Program = Moldo^j

fk'irab|9flpqob^j;|

fkq|j^fk%&
x
||,|fkmrq
||pqa77'lrq|99|-@l|jmrqb|^5|c|o|^|:|<|-8||
||fkq|^8
||pqa77'fk|;;|^8

||,|'ljmrq^qf|k
||fkq|_|:|^|^|^8|_|_|:|^|/
||_|:|^|^|^8|_|_|_|_|:|^|^|^1

||,||lrqmrq|_|'|_|)|f+b+)|^5
||pqa77'lrq|99|^|99|-[5|:|^|^99|_|'|_|99|^-+Yk-8
||obqrok|-8
z
```

308

Caesar-Code: Entschlüsselung

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

Jetzt: Verschiebung um 3
nach *rechts*

Interessante Art, power8.cpp auszugeben:

■ `./caesar_encrypt < power8.cpp | ./caesar_decrypt`