Definition vs. Deklaration

- Deklaration führt Namen ein
- Manche Deklarationen sind Definitionen
- Definition weist Speicher zu und führt Initialisierung durch, wenn nötig

One Definition Rule (ODR)

- Jede Funktion, Klasse, Variable, … darf nur eine Definition haben
- Wiederholte Deklaration jedoch möglich
- Jedes Programm muss genau eine Definition enthalten für jede *potentiell verwendete* Funktion, Klasse, Variable, …

Gültigkeitsbereiche von Namen (*scope*)

- Durch Deklaration eingeführter Name verwendbar in seinem *scope* (Gültigkeitsbereich)
- Faustregel: ab Deklaration bis Ende des enthaltenden *Blocks* (geschweifte Klammern)
- Funktionsparameter insb. im Körper der Funktion
- *Überschattung*: Deklaration des gleichen Namens in verschaltetem Block
- Überschattung erlaubt, jedoch oft verwirrend – lieber vermeiden

Speicherklassen (*storage duration*)

- *Speicherklasse* bestimmt: Wann Speicher zugewiesen? Wo? Wann wieder freigegeben?
- Angabe bei Deklaration durch Schlüsselwörter
- Voreinstellung: *automatisch*; Speicher zuweisen und Initialisieren bei Ausführung der Definition, auf *stack*, Freigabe am Ende des syn. Gültigkeisbereichs
- `static`: Speicher zuweisen am Start des *Programms*, Initialisierung bei *erster* Ausführung der Definition, Freigabe am Ende des *Programms*
- *dynamisch*: Zuweisen mit `new`, auf dem *heap*, Freigeben mit `delete`

Beispiel: `strtok`

strtok.cpp

```cpp
#include <iostream>
#include <iomanip>

using namespace std;

bool is_delim(char c, const char* del) {
  for (int i = 0; del[i] != '\0'; i++)
    if (del[i] == c)
      return true;
  return false;
}
int strspn(const char* str, const char* del) {
  int res = 0;
  while (str[res] != '\0')
    if (is_delim(str[res], del))
      res++;
    else
      return res;
  return res;
}
```

strcspn

```cpp
char* strtok(char* str, const char* del) {
  static char* buffer;
  if (str)
    buffer = str;

  buffer += strspn(buffer, del);

  if (*buffer == '\0')
    return nullptr;

  char* tokenBegin = buffer;

  buffer += strcspn(buffer, del);
  if (*buffer != '\0')
    *(buffer++) = '\0';

  return tokenBegin;
}

int main() {
  char input[] = "one + two * (three - four)!";
  const char delim[] = "! +- (*)";
  for (char* token = strtok(input, delim); token; token =
  ↪  strtok(nullptr, delim))
    cout << '"' << token << '"' << endl;
}
```

```
"one"
"two"
"three"
"four"
```

Bindungswirkung von Namen (*linkage*)

- Relevant bei Zusammenfügen *mehrerer* Übersetzungseinheiten zu Programm
- Keine Bindungswirkung: Name verweist nur in Gültigkeitsbereich des Namens auf Objekt (Funktion, Variable, Klasse, …)
- Interne Bindung: Name verweist innerhalb von *Übersetzungseinheit* auf gleiches Objekt
- Externe Bindung: Name verweist innerhalb von *Programm* auf gleiches Objekt
- Schlüsselwort `extern` ⟶ Externe Bindung
- `static` ⟶ Interne Bindung
- Deklarationen in *global scope* (außerhalb von Funktionen, Blöcken, etc.) haben externe Bindung
- `const`, `constexpr` ⟶ Voreinstellung interne Bindung
- Klasse vererbt Bindungswirkung an Komponenten

Deklaration von namespaces

[“inline”] “namespace” [⟨Name⟩] “{” {⟨Deklaration⟩} “}”

- Fügt enthaltene Deklarationen zu *Namensraum* hinzu
- Zugriff dann mit scope resolution operator `::`
- “`inline`” macht Namen *auch* im aktuellen scope bekannt (verschachtelte namespaces)
- Namespaces (und enthaltene Deklarationen) haben externe Bindung

Unbenannte namespaces

- Namespace Deklaration ohne Name agiert wie eine mit eindeutigem Namen *pro Übersetzungseinheit*
- Unbenannte namespaces (und enthaltene Deklarationen) haben interne Bindung
- Unbenannte namespaces verhalten sich als wären sie `inline`

Namespace alias

"namespace" ⟨Name⟩ "=" ⟨Name⟩ ";"

- Führt weiteren Namen ein für bereits existenten namespace