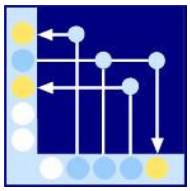
Hochschule Aalen
Fakultät Elektronik und Informatik
Studienbereich Informatik



Programmieren in C++

Vorlesung im Sommersemester 2022
Prof. Dr. habil. Christian Heinlein

8. Übungsblatt (23. Mai 2022)

Aufgabe 11: Template-Metaprogrammierung

a) Definieren Sie eine rekursive Typschablone `Bin<N, K>` mit geeigneten Spezialisierungen, sodass das statische konstante Datenelement `Bin<N, K>::value` für $N = 0, 1, \ldots$ und $K = 0, \ldots, N$ jeweils den zur Übersetzungszeit berechneten Wert des Binomialkoeffizienten

$$\binom{N}{K} = \begin{cases} 1 & \text{für } K = 0 \text{ oder } K = N \\ \binom{N-1}{K-1} + \binom{N-1}{K} & \text{sonst} \end{cases}$$

enthält!

✎

```
// Allgemeine rekursive Typschablone zur Berechnung von N über K.
template <int N, int K>
struct Bin {
    static const int value = Bin<N-1, K-1>::value + Bin<N-1, K>::value;
};

// Partielle Spezialisierung für N über 0.
template <int N>
struct Bin <N, 0> {
    static const int value = 1;
};

// Partielle Spezialisierung für N über N.
template <int N>
struct Bin <N, N> {
    static const int value = 1;
};
```

```
// Explizite Spezialisierung für 0 über 0, weil in diesem Fall
// beide partiellen Spezialisierungen gleich gut passen und
// Bin<0, 0> sonst mehrdeutig wäre.
template <>
struct Bin <0, 0> {
    static const int value = 1;
};
```

✎

b) Definieren Sie – ohne Verwendung der obigen Typschablone – eine rekursive Variablenschablone `bin<N, K>` mit geeigneten Spezialisierungen, sodass `bin<N, K>` ebenfalls den zur Übersetzungszeit berechneten Wert des Binomialkoeffizienten $\binom{N}{K}$ enthält!

✎

```
// Allgemeine rekursive Variablenschablone zur Berechnung von N über K.
template <int N, int K>
const int bin = bin<N-1, K-1> + bin<N-1, K>;

// Partielle Spezialisierung für N über 0.
template <int N>
const int bin<N, 0> = 1;

// Partielle Spezialisierung für N über N.
template <int N>
const int bin<N, N> = 1;

// Explizite Spezialisierung für 0 über 0.
template <>
const int bin<0, 0> = 1;
```

✎

c) Definieren Sie Variablenschablonen mit geeigneten Spezialisierungen mit folgender Bedeutung (alle Schablonenparameter bezeichnen ganze Zahlen):

- `is_divisible<m, n>` ist genau dann `true`, wenn `m` durch `n` teilbar ist.
- `is_divisible_range<m, n>` ist genau dann `true`, wenn `m` durch eine Zahl von 2 bis `n` teilbar ist.
- `is_prime<m>` ist genau dann `true`, wenn `m` eine Primzahl ist.
  Hierfür kann z. B. überprüft werden, ob `m` durch eine Zahl von 2 bis `m/2` teilbar ist.
- `next_prime<m>` ist die kleinste Primzahl, die größer oder gleich `m` ist.

  Mögliche Verwendung:

  ```
  // Streuwerttabelle, deren Größe die nächste Primzahl nach 1000 ist.
  string hashtab [next_prime<1000>];
  ```

- `prime<i>` ist die `i`-te Primzahl, d. h. `prime<0>` ist 2, `prime<1>` ist 3 usw.
- `is_divisible_prime<m, n>` ist genau dann `true`, wenn `m` durch eine Primzahl von 2 bis `n` teilbar ist.

  Das ist mathematisch gleichbedeutend mit `is_divisible_range<m, n>`, kann aber effizienter implementiert werden, indem nur die Teilbarkeit von `m` durch Primzahlen überprüft wird. Damit kann dann wiederum `is_prime` effizienter implementiert werden.

Hinweise:

- Bei Bedarf können die Schablonen zusätzliche optionale Parameter besitzen, anhand derer dann geeignete Spezialisierungen definiert werden können (vgl. §5.2.4 der Vorlesungsfolien).
- Wenn die Definitionen mehrerer Variablenschablonen wechselseitig voneinander abhängen, kann eine von ihnen durch Angabe des Schlüsselworts `extern` vor ihrem Typ (wie bei gewöhnlichen Variablen) zunächst nur deklariert, aber noch nicht definiert werden.

✎

```
// Ist m durch n teilbar?
template <int m, int n>
const bool is_divisible = m % n == 0;

// Ist m durch eine Zahl von 2 bis n teilbar?
template <int m, int n>
const bool is_divisible_range =
            is_divisible<m, n> || is_divisible_range<m, n - 1>;

// Spezialisierung/Rekursionsabbruch mit Wert false (!) für n = 1.
template <int m>
const bool is_divisible_range<m, 1> = false;

// Ist m eine Primzahl?
#ifdef SIMPLE
// Einfache Implementierung, wenn Makro SIMPLE definiert ist.
template <int m>
const bool is_prime = !is_divisible_range<m, m/2>;
#else
template <int m>
extern const bool is_prime;
#endif

// Die kleinste Primzahl >= m.
template <int m, bool = is_prime<m>>
const int next_prime;

// Spezialisierung für Primzahlen m.
template <int m>
const int next_prime<m, true> = m;

// Spezialisierung für Nicht-Primzahlen m.
template <int m>
const int next_prime<m, false> = next_prime<m + 1>;

// Die i-te Primzahl.
template <int i>
const int prime = next_prime<prime<i - 1> + 1>;

// Spezialisierung/Rekursionsabbruch für i = 0.
template <>
const int prime<0> = 2;

// Ist m durch eine Primzahl von prime<i> bis n teilbar?
template <int m, int n, int i = 0, bool = prime<i> <= n>
const bool is_divisible_prime;
```

```
// Spezialisierung für prime<i> <= n.
template <int m, int n, int i>
const bool is_divisible_prime<m, n, i, true> =
    is_divisible<m, prime<i>> || is_divisible_prime<m, n, i + 1>;

// Spezialisierung/Rekursionsabbruch mit Wert false für prime<i> > n.
template <int m, int n, int i>
const bool is_divisible_prime<m, n, i, false> = false;

// Ist m eine Primzahl?
#ifndef SIMPLE
// Effizientere Implementierung, wenn Makro SIMPLE nicht definiert ist.
template <int m>
const bool is_prime = !is_divisible_prime<m, m/2>;
#endif
```

✎

Aufgabe 12: Untersuchung von Typen

Definieren Sie geeignete Schablonen sowie eventuell erforderliche Hilfsschablonen mit folgender Bedeutung:

a) Für einen beliebigen Typ `T` ist `pointer_depth<T>` die Anzahl der „Zeigerebenen“ und `pointer_base<T>` der ggf. indirekte „Basistyp“ von `T`, zum Beispiel:

T	pointer_depth<T>	pointer_base<T>
int	0	int
int*	1	int
const int * volatile * const	2	const int
int* ()	0	int* ()
int (*) ()	1	int ()

Der Typ `int* ()` ist ein Funktionstyp mit Resultattyp `int*`; deshalb ist `pointer_depth<int* ()>` gleich 0, obwohl der Resultattyp des Funktionstyps ein Zeigertyp ist.
`int (*) ()` hingegen ist ein Funktionszeigertyp mit Basistyp `int ()`.

✎

```
// Hilfstyp.
template <typename T>
struct pointer_help {
    // Sofern durch eine Spezialisierung nichts anderes definiert wird,
    // ist die Anzahl der Zeigerebenen von T gleich 0
    // und der Basistyp von T gleich T.
    static const int depth = 0;
    typedef T type;
};
```

```
// Spezialisierung für unqualifizierte Zeigertypen.
template <typename T>
struct pointer_help <T*> {
    // Die Anzahl der Zeigerebenen von T* ist die Anzahl der
    // Zeigerebenen von T plus 1.
    // Der Basistyp von T* ist gleich dem Basistyp von T.
    static const int depth = pointer_help<T>::depth + 1;
    typedef typename pointer_help<T>::type type;
};

// Spezialisierungen für const- und/oder volatile-qualifizierte
// Zeigertypen.

template <typename T>
struct pointer_help <T* const> : pointer_help<T*> {};

template <typename T>
struct pointer_help <T* volatile> : pointer_help<T*> {};

template <typename T>
struct pointer_help <T* const volatile> : pointer_help<T*> {};

// Schablonen zur angenehmeren Verwendung.

template <typename T>
const int pointer_depth = pointer_help<T>::depth;

template <typename T>
using pointer_base = typename pointer_help<T>::type;
```

✎

b) Für einen beliebigen Typ `T` ist `is_func<T>` genau dann `true`, wenn `T` ein Funktionstyp ist.
In diesem Fall ist `result_type<T>` der Resultattyp und `param_count<T>` die Anzahl der Parameter des Funktionstyps `T`. Andernfalls sind `result_type<T>` und `param_count<T>` nicht definiert.

✎

```
// Hilfstyp.
template <typename T>
struct func_help {
    // Sofern durch eine Spezialisierung nichts anderes definiert wird,
    // ist T kein Funktionstyp, und dementsprechend gibt es weder
    // Resultattyp noch Parameterzahl.
    static const bool flag = false;
};
```

```
// Spezialisierung für Funktionstypen.
template <typename R, typename ... PP>
struct func_help <R (PP ...)> {
    // R (PP ...) ist ein Funktionstyp mit Resultattyp R
    // und Parameterzahl sizeof...(PP).
    static const bool flag = true;
    typedef R type;
    static const int count = sizeof...(PP);
};

// Schablonen zur angenehmeren Verwendung.

template <typename T>
const bool is_func = func_help<T>::flag;

template <typename T>
using result_type = typename func_help<T>::type;

template <typename T>
const int param_count = func_help<T>::count;
```

Beachten Sie die Erläuterung zum Schlüsselwort `typename` in §5.3.3 der Vorlesungsfolien!