1 Fibonacci

1.1 Aufgabe

In dieser Aufgabe wollen wir das Typ-System für Berechnungen zur Compiletime verwenden.

Schreiben Sie eine Klasse, welche mithilfe *rekursiver Templates* die n-te Fibonacci-Zahl berechnet und folgendermassen aufgerufen wird:

```
1   constexpr auto value = Fibonacci<16>::value;
2   std::cout << value << std::endl;
```

2 Retro zu Modern

Der folgende Code für die Berechnung von 2^e zur Compiletime ist vorgegeben.

```
1  template<int e>
2  struct TwoExponential {
3    enum { value = 2 * TwoExponential<e - 1>::value };
4  };
5
6  template<>
7  struct TwoExponential<0> {
8    enum { value = 1 };
9  };
```

Entwickeln Sie diesen Code in drei Schritten weiter, damit alle folgenden vier Aufrufe gültig sind.

```
1   std::cout << "2^4 = " << TwoExponential<4>::value << std::endl;
2   std::cout << "2^4 = " << Exponential<2, 4>::value << std::endl;
3   std::cout << "2^4 = " << ExponentialDerived_v<2, 4> << std::endl;
4   std::cout << "2^4 = " << ExponentialExpr(2, 4) << std::endl;
```

2.1 Aufgabe

a) Schreiben Sie eine Variante `Exponential` welche immer noch Rekursion verwendet. Ersetzen Sie aber den Enum-Hack und erweitern Sie die Funktion, dass auch die Basis frei gewählt werden kann: x^e.

b) Schreiben Sie eine Variante `ExponentialDerived` die `std::integral_constant` verwendet. Zudem soll diese einen Helper `ExponentialDerived_v` anbieten, welcher die Verwendung vereinfacht.

c) Zum Schluss konvertieren Sie die Funktion in eine `constexpr`-Funktion `ExponentialExpr`, welche Meta-Programming ohne jegliche Templates erlaubt.
d) Testen Sie zudem, ob alle ihre Implementierungen wirklich zur Compiletime evaluiert werden. Welches Keyword wird dazu benötigt?

3 Type-Conditional

Schreiben Sie ein Programm mit Templates, welches folgende Funktionalität aufweist:

```
1   std::cout << sizeof(BiggerType<uint64_t, uint32_t>) << std::endl;   // 8
2   std::cout << sizeof(SmallerType<uint64_t, uint32_t>) << std::endl;  // 4
```

3.1 Aufgabe

Verwenden Sie Template-Spezialisierungen und Alias Templates.

4 Typ-Auswahl für Bit-Count

In dieser Aufgabe soll ein Mechnismus entwickelt werden, welcher ihnen zu einer vorgegebenen Anzahl Bits einen passenden *unsigned Integer* Datentyp zurückgibt. Am Ende soll der Code folgendermassen geprüft werden können:

```
1   static_assert(std::is_same_v<bool, IntegerForBits_t<1>>);
2   static_assert(std::is_same_v<uint8_t, IntegerForBits_t<2>>);
3   static_assert(std::is_same_v<uint8_t, IntegerForBits_t<8>>);
4   static_assert(std::is_same_v<uint16_t, IntegerForBits_t<9>>);
5   static_assert(std::is_same_v<uint16_t, IntegerForBits_t<16>>);
6   static_assert(std::is_same_v<uint32_t, IntegerForBits_t<17>>);
7   static_assert(std::is_same_v<uint32_t, IntegerForBits_t<32>>);
8   static_assert(std::is_same_v<uint64_t, IntegerForBits_t<33>>);
9   static_assert(std::is_same_v<uint64_t, IntegerForBits_t<64>>);
```

4.1 Aufgabe

- Schreiben Sie eine Funktion `ByteCount(size_t bit_count)`, welche zur Compiletime berechnen kann, wie viele Bytes mindestens benötigt werden, um die vorgegebene Anzahl Bits (`bit_count`) zu speichern.

- Erstellen Sie einen Mechanismus `ByteType`, welcher anhand einer vorgegebenen Anzahl Bytes einen passenden *unsigned Integer* Datentyp auswählt, welcher mindestens benötigt wird, um alle Bytes zu speichern. Verwenden Sie dazu *template specialization*. Denken Sie auch an den Fall, wenn die Anzahl Bytes zu gross für einen der existierenden primitiven Integer Typen ist.
- Erstellen Sie den Mechanismus `IntegerForBits`, welcher die beiden bereits erstellten Teile kombiniert, um einen Typ entsprechend der Anzahl vorgegebener Bits zu wählen. Dieser soll auch den Spezialfall von einem Bit handhaben, bei welchem der Typ `bool` gewählt werden soll.
- Damit Sie das Ganze einfach benutzten können, schreiben Sie noch einen Helper `IntegerForBits_t`, welcher den ausgewählten Typ direkt zurückgibt.