1. Template (sablon)

1.1. Függvénysablon

Maximum függvény megvalósítása függvénynév túlterheléssel.

```
inline float Max ( float a, float b ) {
   return a>b ? a : b;
}
inline double Max ( double a, double b ) {
   return a>b ? a : b;
}
inline UT1 Max ( UT1 a, UT1 b ) {
   return a>b ? a : b;
}
```

Az unalmas munkát hagyjuk a fordítóra, megvalósítás sablonnal.

```
template < typename T >
T Max ( T a, T b ) { return a>b ? a : b; }
```

1.2. Függvénysablon példányosítás

`Max` sablon példányosítása. Sablon paramétert a fordító határozza meg a függvény paraméter típusa alapján.

```
double x, y, res;
// ...  assigning some values to x & y
res = Max(x-1,y+2.5); // Max<double>
int m = Max( 5, 13);  //  Max<int>
```

Ha nincs függvényparaméter, akkor hogyan lehet példányosítani a függvényt?

```
template < typename T >
T any ( void ) {
   // return a random value of type T;
}
```

Példányosítás explicit minősítővel.

```
int i = any();       // Hibás
int i = any<int>(); // OK
```

1.3. Osztálysablon

Egyszerű egészeket tartalmazó stack osztály.

```
class Stack {
  // implementation
  int top;
  int S[100];
public:
  // interface
  Stack() : top(-1) { }
  void push ( int V ) { S[++top] = V; }
  int pop (void) { top--; return S[top+1]; }
  // other operations
  . . .
};
```

Írjunk generikus stack osztályt, tetszőleges típusú adatok tárolására. A `T` típustól bizonyos operátorokat elvárunk, pl. =.

```
template < typename T >
class Stack {
  // implementation
  int top;
  T S[100];
public:
  // interface
  Stack() : top(-1) { }
  void push ( const T& V ) { S[++top] = V; }
  T pop (void) { top--; return S[top+1]; }
  // other operations
  . . .
};
```

1.4. Osztálysablon példányosítása

```
Stack<int> istk;
Stack<double> dstk;
```

Sablon paraméter egész érték is lehet. Adjuk meg a stack méretét sablon paraméteren keresztül.

```
template <typename T, int N >
class Stack {
  // implementation
  int top;
  T S[N];
public:
  // interface
  Stack() : top(-1) { }
  void push ( const T& V ) { S[++top] = V; }
  T pop (void) { top--; return S[top+1]; }
     // other operations
     . . .
};

Stack<string, 100> sstk;
```

1.5. sablonok definiálása

Szokásos sablon definiálás.

```
template<typename T>
class C {
public:
  C() { . . . }  // konstruktor név nem típus
  ~C() { . . . } // destruktor
  void f() { . . . } // member function
};
```

Sablon definiálás teljes formában

```
template<typename T>
class C {
public:
  C<T>() { . . . }   // konstruktor
  ~C<T>() { . . . }  // destruktor
  void f() { . . . } // member function
};
```

Hatványozó függvénysablon:

```
template < unsigned N, typename T >
T Power ( const T& v ) {
   T res = v;
   for ( int i=1; i<N; i++ )
   res *= v;
   return res;
}

void f() {
  double d1 = Power<5>(1.2);
  double d2 = Power<5,int>(1.2);
  std::cout << d1 << "␣" << d2;
}
```

1.6. Függvénysablon specializáció

```
template<class T>
bool cmp(T const& a, T const& b) { return a < b; }

template<class T>
bool cmp( T* const& a, T* const& b) { return *a < *b; }

bool cmp(const char* a, const char* b) { return strcmp(a, b)<0; }
```

Faktoriális kiszámítása fordítási időben sablon felhasználásával. Nem biztos, hogy hasznos, de jó példa a specializálásra.

```
template<unsigned N>
unsigned long Fact ( void ) {
   return N*Fact<N-1>();
}
template<>
unsigned long Fact<0>( void ) {return 1; }
```

1.7. Osztálysablon specializáció

```
template < typename T >
class C {
    // common implementation
}
template< typename T >
class C<T*> {
   // implementation for the specified subset
}
```

1.8. Sablon paraméterek lehetséges specializálása

const T	konstans típus
T*	pointer
T&	hivatkozás (referencia)
T[integer-constant]	tömb
type (*)(T)	T típusú argumentummal rendelkező függvénypointer.
T(*)()	T visszatérési értékkel rendelkező függvénypointer.
T(*)(T)	T típusú argumentummal és visszatérési értékkel rendelkező függvénypointer.

1.9. Typename kulcsszó használata

```
template < typename T >
void f () {
  typename T::t1* m2; // declaration
  T::t2* m3;          // expression
}
```

1.10. Függvény objektum

```
template <typename T>
class Greater {
T value;
public:
   Greater(const T& v) : value(v) {}
   bool operator()(const T& x)const { return x>value; } // inline
};

template < typename T, typename Comparator >
T* find ( T* pool, int n, const Comparator& comp ) {
  T* p = pool;
  for ( int i = 0; i<n; i++ ) {
     if ( comp(*p) ) return p; // success
     p++;
  }
  return 0; // fail
}

double A[100]; double* p = find(A, 100, Greater<double>(5) );
```

1.11. Alapadatok inicializálása

```
template<template<typename,int> class Tomb, typename T, int N >
ostream& operator<<(ostream& os, const Tomb<T,N>& tt ) {
  for(int i=0; i<N; i++) cout<< tt.t[i] <<' ';
  return os;
}

template<class T, int N>
class Tomb {
  T t[N];
public:
  Tomb() {
        for (int i = 0; i < N; i++)
            t[i] = T();      // Generikus default. Alaptípusoknál 0
  }

  T& operator[](int i) {
      if (i < 0 || i >= N)
          throw "Index hiba";
      return t[i];
  }

  friend ostream& operator<< <>(ostream& os, const Tomb<T,N>& tt );
};
```

Figyeljük meg az `ostream& operator<< <>(....)` deklarációjánál a <> üres template paramétert. Mivel a barát függvényt függvénysablonból állítjuk elő, ezért ezt jelezni kell. Ha a függvény paraméterből a fordító következtetni tud a sablon paraméterre, akkor a sablon paraméter lehet üres is. Ha nem írjuk ki, akkor a fordító automatikusan nem példányosítja a barát operátort és linkelési hiba keletkezik.

A `Tomb()` konstruktorban a t[i] = T() értékadás a beépített típusokat inicializálja. Ha ez nincs, és `int` típusú adatokat tárolunk a tömbbe, akkor az elemek értéke nem meghatározott. Osztályok esetén viszont a tömb elemeit kétszer inicializáljuk. Hogyan lehet ezt elkerülni?

Módosítsuk a konstruktort a következő módon:

```
template<class T, int N> class Tomb {
  T t[N];
public:
  Tomb() {
      if( !IsClassT<T>::Yes ) {
        for (int i = 0; i < N; i++)
            t[i] = T();       // Generikus default. Alaptípusoknál 0
      }
  }
  .......
```

Írjuk meg a `IsClassT<T>` osztálysablont.

```
template< typename T>
class IsClassT {
  public:
   template<class X> static char Test( int X::* );
   template<class X> static long Test( ... );
   enum { Yes = sizeof( IsClassT<T>::Test<T>( 0 ) ) == 1 };
   enum { No = !Yes };
};
```

A megoldáshoz részleges tagfüggvénysablon specializálást használhatunk. A `Test()` függvény különböző méretű adatot ad vissza, annak a függvényében, hogy milyen paraméterrel hívjuk. `::` tagválasztó operátora csak osztályoknak (struktúráknak) lehet. A `Test(int X::* )` egész tagváltozóra mutató pointert vár. Egyébként a `Test(...)` függvény játszik szerepet. Az `IsClassT<T>::Test<T>(0)` paramétere trükkösen 0, ami lehet egy memóriacím érték, de illeszkedik a változó függvényparaméter listára is. Ha a `T` sablon paraméter `class`, akkor a `char Test(int X::*)` függvényt generálja ki a fordító. A `sizeof()` operátor fordítási időben értékelődik ki, tehát csak az fontos, hogy a `T` sablon paraméterrel melyik tagfüggvényt példányosítja a fordító, de a függvény futási időben nem hívódik meg. Ezért a `Test()` függvényeket csak deklaráltuk, és nem definiáltuk. Akkor is a `char Test(int X::*)` függvényt példányosítja a fordító, ha olyan osztállyal példányosítjuk a sablont, aminek egyetlen tagváltozója sincs.

Sajnos ezt a szép megoldást az általunk ismert fordítók nem értik, fordítási hiba keletkezik. Az `IsClassTHelper` belső struktúra bevezetésével azonban elegendő segítséget kapnak a fordítók, így már megbírkoznak a feladattal Az IsClassT template a `www.hit.bme.hu/~izso/modtomb.cpp` példaprogram segítségével tesztelhető.

```
template<typename T> class IsClassT {

    struct IsClassTHelper
    {
        template<typename X> static char Test( int X::* );
        template<typename X> static long Test( ... );
    };

    typedef  IsClassTHelper H;

public:
    enum { Yes = sizeof( H::template Test<T>( 0 ) ) == sizeof(char) };
    enum { No = !Yes };
};
```

Írjunk osztálysablont, ami eldönti, hogy az első paramétere konvertálható-e a második paraméterre.

```
template<class T, class U> class Conversion {
   typedef char Small;
   struct Big { char dummy[4]; };
   static Small Test(U);
   static Big   Test(...);
   static T MakeT();
public:
   enum{ exists = sizeof( Test( MakeT() ))==sizeof( Small ) };
};
```