```cpp
#include <iostream>
#include <string>

using namespace std;

 void my_swap (int &a, int &b )
 {
     int tmp = a;
     a=b;
     b=tmp;
 }

 void my_swap (string &a, string &b )
 {
     string tmp = a;
     a=b;
     b=tmp;
 }

int main()
{
   string a, b;
   a = "hello"; b = "world";
   cout << "before a = " << a << " b = " << b << endl;
   my_swap (a,b);
   cout << "after  a = " << a << " b = " << b << endl;

   int x, y;
   x = 33; y = 44;
   cout << "before x = " << x << " y = " << y << endl;
   my_swap(x,y);
   cout << "before x = " << x << " y = " << y << endl;

   double d1, d2;
   d1 = 3.3; d2 = 4.4;
   cout << "before d1 = " << d1 << " d1 = " << d2 << endl;
   // my_swap(d1,d2);  // compile time error "no know conversion from double to &int ...
   cout << "before d1 = " << d1 << " d2 = " << d2 << endl;

   return 0;
}
```

../GC01–overloading.cpp

```cpp
#include <iostream>
#include <string>

using namespace std;

 void my_swap (void* &a, void* &b )
 {
     void* tmp = a;
     a=b;
     b=tmp;
 }

int main()
{
   void* a; void* b;
   a = new std::string("hello"); b = new std::string("world");
   cout << "before a = " << *((string*) a) << " b = " << *((string*) b) << endl;
   my_swap (a,b);
   cout << "after  a = " << *((string*) a) << " b = " << *((string*) b) << endl;

   void* x; void* y;
   x = new int(33); y = new int(44);
   cout << "before x = " << *((int*) x) << " y = " << *((int*) y) << endl;
   my_swap(x,y);
   cout << "after x = " << *((int*) x) << " y = " << *((int*) y) << endl;

   cout << "a = " << *((int*) a) << endl; // no compile time error, no runtime error
                    // output a = 1919907594   :(

   return 0;
}
```

../GC02a–voidPointer.cpp

```cpp
#include <iostream>
#include <string>

using namespace std;

 void my_swap (void* &a, void* &b )
 {
     void* tmp = a;
     a=b;
     b=tmp;
 }

int main()
{
   void* a; void* b;
   a = new std::string("hello"); b = new std::string("world");
   cout << "before a = " << *(static_cast<string*>(a)) << " b = " << *(static_cast<string*>(b)) << endl;
   my_swap (a,b);
   cout << "after  a = " << *(static_cast<string*>(a)) << " b = " << *(static_cast<string*>(b)) << endl;

   void* x; void* y;
   x = new int(33); y = new int(44);
   cout << "before x = " << *(static_cast<int*>(x)) << " y = " << *(static_cast<int*>(y)) << endl;
   my_swap(x,y);
   cout << "after x = " << *(static_cast<int*>(x)) << " y = " << *(static_cast<int*>(y)) << endl;

   cout << "a = " << *(static_cast<int*>(a)) << endl; // no compile time error, no runtime error
                            // output a = 1919907594   :(

   return 0;
}
```

../GC02b–voidPointer.cpp

```cpp
/*
 * centralElementTemplate.cpp
 *
 * alberto.ferrari@unipr.it
 *
 * Example: function with template
 *
 */

#include <iostream>
#include <string>
using std::cout;
using std::endl;
using std::string;

template <typename T>
T centralElement(T data[], int cont)
{
    return data[cont/2];
}

int main( )
{
  int i[] = {10,20,30,40,50};
  int ci = centralElement(i,5);
  cout<< ci << endl;

  string s[] = {"alpha","beta","gamma"};
  string cs = centralElement(s,3);
  cout<< cs << endl;

  float f[] = {2.2,3.3,4.4};
  float cf = centralElement<float>(f,3);
  cout<< cf << endl;
}
```

../GC03b–centralElementTemplate.cpp

```cpp
/*
 * funTemplateExplicitOverloading.cpp
 *
 * alberto.ferrari@unipr.it
 *
 * Example: function with template
 *
 */

#include <iostream>
#include <string>
using std::cout;
using std::endl;
using std::string;

template<class T>
T square(T b)
{
      return b * b;
}

template<>
string square(string b)
{
  return b + b;
}

int main( )
{
    int i = 5;
    cout << "square "<< i << " = " << square(i) << endl;

    double j = 5.5;
    cout << "square "<< j << " = " << square(j) << endl;

  string s = "hello";
  cout << "square "<< s << " = " << square(s) << endl;

  char c = 'h';
  cout << "square "<< c << " = " << square(c) << endl;
}
```

../GC03b–ExplicitOverloading.cpp

```cpp
#include <iostream>
#include <string>

using namespace std;

template <class T>
void my_swap(T& f, T& s) {
    T tmp = f;
    f = s;
    s = tmp;
}

int main()
{
   int a = 3; int b = 4;
   cout << "before a = " << a << " b = " << b << endl;
   my_swap<int> (a,b);
   cout << "after  a = " << a << " b = " << b << endl;

   string s1 = "hello";
   string s2 = "world";
   cout << "before s1 = " << s1 << " s2 = " << s2 << endl;
   my_swap<string> (s1,s2);
   cout << "after  s1 = " << s1 << " s2 = " << s2 << endl;

   return 0;
}
```

../GC03–swapTemplate.cpp

```cpp
#include <iostream>
#include <string>

using namespace std;

template<class T>
constexpr T pi = T(3.1415926535897932385);  // variable template

template<class T>
T circular_area(T r) // function template
{
    return pi<T> * r * r; // pi<T> is a variable template instantiation
}

int main() {
  double r1 = 10.0;
  cout << circular_area<double>(r1) << endl;

  int r2 = 10;
  cout<< circular_area<int>(r2) << endl;

}
```

../GC04–varTemplate.cpp

```cpp
#include <iostream>

template <unsigned int n>
struct factorial {
  const static int value = n * factorial<n - 1>::value ;
};

template <>
struct factorial<0> {
  const static int value = 1;
};

int main() {
  std::cout << "5! = " << factorial<5>::value << std::endl;
  std::cout << "0! = " << factorial<0>::value << std::endl;
  return 0;
}
```

../GC05b–factorial.cpp

```cpp
#include <iostream>

using namespace std;

// Recursive template for general case
template <int N>
struct factorial {
  enum { value = N * factorial<N - 1>::value };
};

// Template specialization for base case
template <>
struct factorial<0> {
  enum { value = 1 };
};

int main() {
    cout << factorial<5>::value;
}
```

../GC05–factorial.cpp

```
/*
 * minValueTemplate.cpp
 *
 * alberto.ferrari@unipr.it
 *
 * Example: function with template
 *
 */

#include <iostream>
using std::cout;
using std::endl;

class Point
{
public:
   Point();
   Point(int, int);
   ~Point();
   void setX(int);
   int getX();
   void setY(int);
   int getY();
   void display();
private:
   int x;
   int y;
};

Point::Point ()
{
   x = 0;
   y = 0;
}

Point::Point (int vx, int vy)
{
  x = vx;
  y = vy;
}
Point::~Point()
{
}

void Point::setX (int vx)
{
  x=vx;
}
int Point::getX()
{
   return x;
}
void Point::setY (int vy)
{
  y=vy;
}
int Point::getY()
{
   return y;
}
void Point::display()
{
  cout<<"("<<x<<","<<y<<")"<<endl;
}

template <typename T>
T minValue(T v1, T v2)
{
    if (v1<v2)
        return v1;
    return v2;
}

int main( )
{
```

```
  int mi = minValue(3,6); //(int int) OK
  cout<<mi<<endl;

  float mf1 = minValue(9.2,6.1); // (float float) OK
  cout<<mf1<<endl;

  //float mf2 = minValue(9.2,6); // (float int) template argument deduction/substitution failed
  //cout<<mf2<<endl;

  float mf3 = minValue<float>(9.2,6); //explicit provide type parameter OK
  cout<<mf3<<endl;

  Point p1(3,4);
  p1.display();
  Point p2(5,2);
  p2.display();
  // Point p3 = minValue(p1,p2); //error no match for 'operator<' (operand types are 'Point' and 'Point')

}
```

../GC06–Template–Point.cpp

```cpp
#include <iostream>
#include <string>

using std::cout;
using std::endl;
using std::string;

template <typename F, typename S>
class Pair
{
public:
    Pair(const F& f, const S& s);
    F get_first() const;
    S get_second() const;
private:
    F first;
    S second;
};

template <typename F, typename S>
Pair<F,S>::Pair(const F& f, const S& s)
{
    first = f;
    second = s;
};

template <typename F, typename S>
F Pair<F,S>::get_first() const
{
    return first;
};

template <typename F, typename S>
S Pair<F,S>::get_second() const
{
    return second;
};

int main( )
{
  Pair<int,double> p1(2,3.4);
  int p1_first = p1.get_first();
  double p1_second = p1.get_second();
  cout<<p1_first<<" "<<p1_second<<endl;


  Pair<string,int> p2("alpha",5);
  string p2_first = p2.get_first();
  int p2_second = p2.get_second();
  cout<<p2_first<<" "<<p2_second<<endl;

  Pair<string,string> p3("hello","world");
  string p3_first = p3.get_first();
  string p3_second = p3.get_second();
  cout<<p3_first<<" "<<p3_second<<endl;

}
```

../GC07–pairTemplateClass.cpp

```cpp
#include <string>
#include <locale>
using namespace std::literals;

// Declaration of the concept "EqualityComparable", which is satisfied by
// any type T such that for values a and b of type T,
// the expression a==b compiles and its result is convertible to bool
template<typename T>
concept bool EqualityComparable = requires(T a, T b) {
    { a == b } -> bool;
};

void f(EqualityComparable&&); // declaration of a constrained function template
// template<typename T>
// void f(T&&) requires EqualityComparable<T>; // long form of the same

int main() {
  f("abc"); // OK, std::string is EqualityComparable
  f(std::use_facet<std::ctype<char>>(std::locale{})); // Error: not EqualityComparable
}
```

../EqualityComparable.cpp

```cpp
/*
 * minValueTemplate.cpp
 *
 * alberto.ferrari@unipr.it
 *
 * Example: function with template
 *
 */

#include <iostream>
#include <string>
using std::cout;
using std::endl;
using std::string;

class Point
{
public:
   Point();
   Point(int, int);
   ~Point();
   void setX(int);
   int getX();
   void setY(int);
   int getY();
   void display();
private:
   int x;
   int y;
};

template <typename T>
concept bool Equality_Comparable()
{
    return requires(T a, T b) {
        {a == b} -> bool;
        {a != b} -> bool;
    };
}

template <Equality_Comparable T>
bool twoEquals(T v1, T v2, T v3)
{
    if (v1==v2 || v1==v3 || v2==v3)
        return true;
    return false;
}

int main( )
{
  cout<<twoEquals(2,3,2)<<endl;      //(int int int) OK
  cout<<twoEquals(9.2,6.1,5.8)<<endl;   //(float float) OK
  cout<<twoEquals(2,3.1,2)<<endl;     //(int float int) ERROR
  cout<<twoEquals<float>(9.2,6,6)<<endl;  //explicit provide type parameter OK
  cout<<twoEquals("alpha","beta","beta")<<endl; //(string string string) OK
  Point p1(3,4);
  Point p2(5,2);
  Point p3(3,4);
  cout<<twoEquals(p1,p2,p3)<<endl;
  // error: cannot call function   bool twoEquals(T, T, T) [with T = Point]
  //error no match for 'operator<' (operand types are 'Point' and 'Point')
  // note:   constraints not satisfied
  // bool twoEquals(T v1, T v2, T v3)
  // ^~~~~~~~~~~~~
  // note: within  template<class T> concept bool Equality_Comparable() [with T = Point]
  // concept bool Equality_Comparable()
  //              ^~~~~~~~~~~~~~~~~~~
  // note:     with   Point  a
  // note:     with   Point  b
  // note: the required expression    (a == b)    would be ill-formed
  // note: the required expression    (a != b)    would be ill-formed
}

Point::Point ()
{
```

```
    x = 0;
    y = 0;
}

Point::Point (int vx, int vy)
{
  x = vx;
  y = vy;
}
Point::~Point()
{
}

void Point::setX (int vx)
{
  x=vx;
}
int Point::getX()
{
   return x;
}
void Point::setY (int vy)
{
  y=vy;
}
int Point::getY()
{
   return y;
}
void Point::display()
{
  cout<<"("<<x<<","<<y<<")"<<endl;
}
```

../testEqualsWithConcepts.cpp

```cpp
/*
 * minValueTemplate.cpp
 *
 * alberto.ferrari@unipr.it
 *
 * Example: function with template
 *
 */

#include <iostream>
#include <string>
using std::cout;
using std::endl;
using std::string;

template <typename T>
concept bool Arithmetic()
{
    return requires(T a, T b) {
    {a + b} -> T;
  };
}

template <Arithmetic T>
T mySum(T v1, T v2)
{
    return v2+v2;
}

int main( )
{

  int mi = mySum(3,3);  //(int int) OK
  cout<<mi<<endl;

  float mf1 = mySum(9.2,6.1); // (float float) OK
  cout<<mf1<<endl;

  //float mf2 = mySum(9.2,6);
  //note:   template argument deduction/substitution failed:
  //note:   deduced conflicting types for parameter   T   (  double   and   int   )
  //cout<<mf2<<endl;

  float mf3 = mySum<float>(9.2,6); //explicit provide type parameter OK
  cout<<mf3<<endl;

  string s1 = mySum("alpha","beta");
  cout<<s1<<endl;
  /*
  Point p1(3,4);
  p1.display();
  Point p2(5,2);
  p2.display();
  // Point p3 = firstOrSecond(p1,p2); //error no match for 'operator<' (operand types are 'Point' and 'Point')
  // note:   constraints not satisfied
  // T firstOrSecond(T v1, T v2)
  // ^~~~~~~~~~~~~
  // note: within  template<class T> concept bool Equality_Comparable() [with T = Point]
  // concept bool Equality_Comparable()
  //              ^~~~~~~~~~~~~~~~~~~
  // note:     with   Point  a
  // note:     with   Point  b
  // note: the required expression    (a == b)    would be ill-formed
  // note: the required expression    (a != b)    would be ill-formed
  // p3.display();
  * */
}
```

../testSTLConcepts.cpp

```cpp
/*
 * testSumconcepts.cpp
 *
 * alberto.ferrari@unipr.it
 *
 * Example: function with template
 *
 */

#include <iostream>
#include <string>
using std::cout;
using std::endl;
using std::string;

template <typename T>
concept bool AdmitsSum()
{
    return requires(T a, T b) {
    {a + b} -> T;
  };
}

template <AdmitsSum T>
T mySum(T v1, T v2)
{
    return v1+v2;
}

int main( )
{

  int mi = mySum(3,3);  //(int int) OK
  cout<<mi<<endl;

  float mf1 = mySum(9.2,6.1); // (float float) OK
  cout<<mf1<<endl;

  float mf3 = mySum<float>(9.2,6); //explicit provide type parameter OK
  cout<<mf3<<endl;

  // string s1 = mySum("alpha","beta");
  // cout<<s1<<endl;
  // error: cannot call function   T mySum(T, T) [with T = const char*]
  // string s1 = mySum("alpha","beta");
  //                                ^
  // note:   constraints not satisfied
  // T mySum(T v1, T v2)
  // ^~~~~
  // note: within  template<class T> concept bool AdmitsSum() [with T = const char*]
  // concept bool AdmitsSum()
  //              ^~~~~~~~~~
  // note:     with   const char* a
  // note:     with   const char* b
  // note: the required expression    (a + b)    would be ill-formed
}
```

../testSumConcepts.cpp

```cpp
/*
 * testIsIntegral.cpp
 *
 * alberto.ferrari@unipr.it
 *
 * Example:
 *
 */

#include <iostream>
#include <type_traits>
#include <complex>

int main() {
  std::cout << std::boolalpha;
  std::cout << "is_integral:" << std::endl;
  std::cout << "char: " << std::is_integral<char>::value << std::endl;
  std::cout << "int: " << std::is_integral<int>::value << std::endl;
  std::cout << "float: " << std::is_integral<float>::value << std::endl;
  std::cout << "bool: " << std::is_integral<bool>::value << std::endl;

  std::cout << "is_arithmetic:" << std::endl;
  std::cout << "char: " << std::is_arithmetic<char>::value << std::endl;
  std::cout << "float: " << std::is_arithmetic<float>::value << std::endl;
  std::cout << "float*: " << std::is_arithmetic<float*>::value << std::endl;
  std::cout << "complex<double>: " << std::is_arithmetic<std::complex<double>>::value << std::endl;
  return 0;
}
```

../testType_traits.cpp