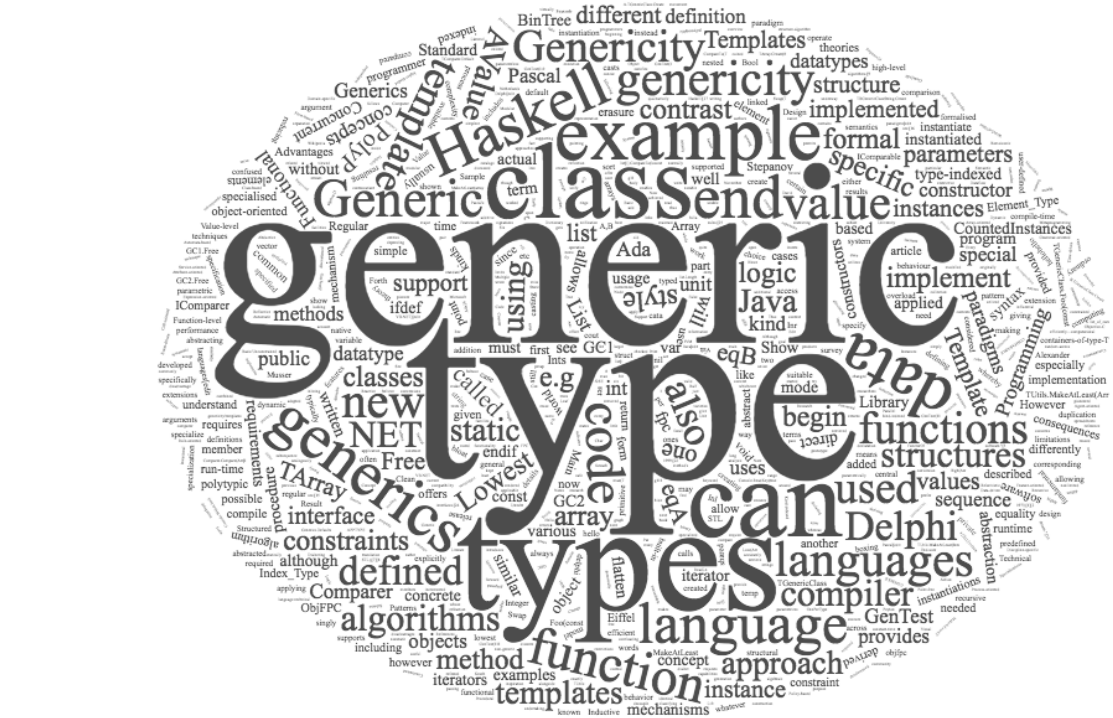
generic programming



alberto ferrari – university of parma

contents

- generic programming in C++
  - function
    - overloading
    - void pointers
    - templates
  - class templates
  - variable templates
- concepts

generic programming in C++

- generic function
  - performs the same operation on different data types
- how to implement a generic function in C++
  - overloading
  - void pointers
  - templates
- example: swap the value of two variables

generic function – overloading

```
void my_swap (int &f, int &s ) {
     int tmp = f; f=s; s=tmp;
}

void my_swap (string &f, string &s ) {
     string tmp = f; f=s; s=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 << "after x = " << x << " by = " << 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 << "after d1 = " << d1 << " d2 = " << d2 << endl;
   return 0;
}
```

overloading: set of methods all having

- the same name
- different arguments list (signature)

generic function – void pointers

```
void my_swap (void* &f, void* &s ) {
     void* tmp = f;
     f=s;
     s=tmp;
 }

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

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

   cout << "a = " << *((int*) a) << endl;
     // no compile time error, no runtime error
     // output a = 1919907594   :(
   return 0;
}
```

- we can write a method that takes a **void pointer** as an argument, and then use that method with any pointer
- this method is more general and can be used in more places
- we need **cast** from void pointer to a specific pointer

generic function – void pointers – static cast

```
void my_swap (void* &f, void* &s ) {
     void* tmp = f;
     f=s;
     s=tmp;
 }

int main() {
   void* a; void* b;
   a = new string("hello"); b = new string("world");
   cout << *(static_cast<string*>(a)) << *(static_cast<string*>(b)) << endl;
   swap (a,b);
   cout << *(static_cast<string*>(a)) << *(static_cast<string*>(b)) << endl;

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

   cout << "a = " << *(static_cast<int*>(a)) << endl;
     // no compile time error, no runtime error
     // output a = 1919907594   :(
   return 0;
}
```

- the **static cast** performs conversions between compatible types
- it is similar to the C-style cast, but is more restrictive
  - for example: the C-style cast would allow an integer pointer to point to a char

generic function – templates

```
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;
}
```

we add a type parameter to the function

template

- templates allows functions and classes to operate with generic types
- with templates a function or a class can work on many different data types without being rewritten for each one
- the *C++ Standard Library* provides many useful functions within a framework of connected templates
- kinds of templates:
  - **function** templates
  - **class** templates
  - **variable** templates (C++14)

function template

A function template

defines a family of functions

```
template <class identifier> function_declaration;
template <typename identifier> function_declaration;
```

template: array central element

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

T must be a type
- primitive type
- class

```
int i[] = {10,20,30,40,50};
int ci = centralElement(i,5);

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

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

Type parameters are **inferred** from the values in a function invocation

Or **explicitly** passed as type parameter

argument deduction

Type parameters are

- inferred from the values in a function invocation
- or explicitly passed as type parameter

```
template <typename T>
T min (T a, T b) {
   return a < b ? a : b;
}
int main() {
   std::cout << min(3,4); // OK (output 3) 'int', 'int' inferred
   std::cout << min(3.3,4); // compile time error
   // template argument deduction/substitution failed:
   // deduced conflicting types for parameter ‘T’ (‘double’ and ‘int’)
   std::cout << min(3.3,(double)(4)); // OK (output 3.3) 'double', 'double' inferred
   std::cout << min(3.3,static_cast<double>(4)); // OK (output 3.3) 'double', 'double' inferred
   std::cout << min<double>(3.3,4); // OK (output 3.3) 'double' explicitly passed
}
```

multiple type parameters

```
template <typename T1, typename T2>
T1 min (T1 a, T2 b) {
   return a < b ? a : b;
}

int main() {
   std::cout << min(3,4) << std::endl;    // output 3 : 'int', 'int' -> 'int'
   std::cout << min(3.3,4) << std::endl;  // output 3.3 'double', 'int' -> 'double'
   std::cout << min(4, 3.3) << std::endl; // output 3 'int', 'double' -> 'int'
}
```

return type parameter

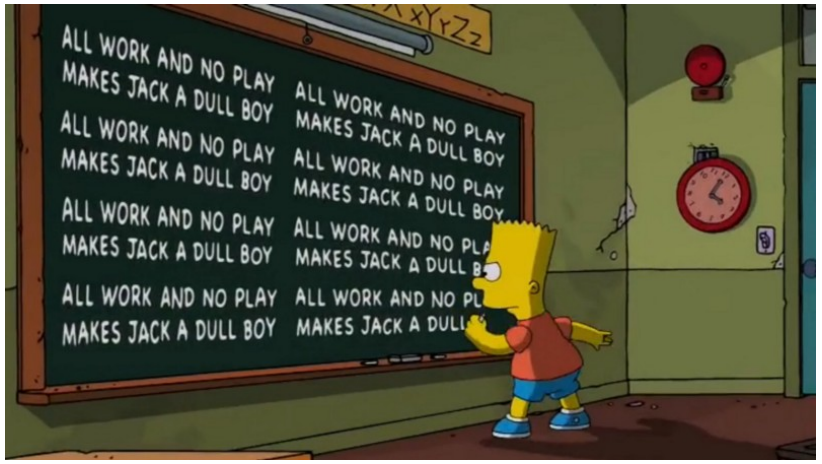
```
template <typename T1, typename T2, typename RT>
RT min (T1 a, T2 b) {
     return static_cast<RT>(a < b ? a : b);
}
int main() {
   std::cout << min<int,int,int>(3,4);
   // output 3 : 'int', 'int' -> 'int'
   std::cout << min<double,int,double>(3.3,4);
   // output 3.3 'double', 'int' -> 'double'
   std::cout << min<int,double,double>(4, 3.3);
   // output 3.3 'int', 'double' -> 'double'
}
```

```
template <typename RT, typename T1, typename T2>
RT min (T1 a, T2 b) {
return static_cast<RT>(a < b ? a : b);
}
int main() {
   std::cout << min<int>(3,4);
   // output 3 : 'int', 'int' -> 'int'
   std::cout << min<double>(3.3,4);
   // output 3.3 'double', 'int' -> 'double'
   std::cout << min<double>(4, 3.3);
   // output 3.3 'int', 'double' -> 'double'
}
```

under the hood

C++ & function template

- in C++, templates are a **pure compile-time feature**
- template is a **factory** that can be used to **produce functions**
- C++ provide **substitutions of types** during compile time
  - in C# substitutions are performed at runtime
- each **set of different template** parameters may cause the generation at compile time of a **different internal function definition**
- the resulting program is bigger in size due to the boilerplate code created during compilation



boilerplate code = sections of code included in many places with little or no alteration

more on template functions

- a **generic function** is also called **template function**
- when the compiler creates a specific version of a generic function, it is said to have created a **generated function**
  - the act of generation is said **instantiation**
- a generated function is a **specific instance** of a template function
- **no code** is generated from a source file that contains only template definitions
- in order for any code to appear, a template must be instantiated → the template arguments must be determined so that the compiler can generate an actual function

generic functions & overloading

- we can define the **explicit** overloading of a generic function
- the modified version overloaded, hide the generic function
- compile-time cases were performed by pattern matching over the template arguments

```
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; //square 5 = 25
    double j = 5.5; cout << "square "<< j << " = " << square(j) << endl; //square 5.5 = 30.25
    string s = "hello";
    cout << "square "<< s << " = " << square(s) << endl; //square hello = hellohello
    char c = 'h';
    cout << "square "<< c << " = " << square(c) << endl; //square h = @
}
```

variable template (c++14)

- a variable template defines a family of variables or static data members
- syntax: `template < parameter-list > variable-declaration`
- example:

```
template<class T>
constexpr T pi = T(3.14159265358979323385);  // 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;   // 314.159
   int r2 = 10;
   cout<< circular_area<int>(r2) << endl;      // 300
   return 0;
}
```

explicit overloading of a variable template

```
template <int N> // Recursive template for general case
struct factorial {
  const static int value = n * factorial<n - 1>::value;
};
template <> // Template specialization for base case
struct factorial<0> {
  const static int value = 1;
};
int main() {
    cout << "4! = " << factorial<4>::value << endl;
}
```

class template

template class can work with many different types of values

class template

- a class template provides a *specification* for ***generating*** classes based on parameters
- class templates are generally used to implement containers
- a class template is *instantiated* by passing a given set of types to it as template ***arguments***

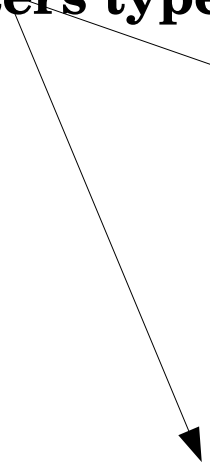
```
template < parameter-list > class-declaration
```

class Pair

```
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;
};
```

class templates unlike function templates

When we declare a variable of a template class you **must specify** the **parameters type.** Types are **not inferred.**



```
Pair<int,double> p1(2,3.4);
int p1_first = p1.get_first();
double p1_second = p1.get_second();
```

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

(C++17 has template deduction of constructors)

templates & compiler

```
template<typename T>
class Foo
{
public:
     T& bar()
     {
         return subject;
     }
private:
     T subject;
};


Foo<int> fooInt;
Foo<double> fooDouble;
```

- The compiler generates the code for the specific types given in the template class instantiation.
- Left side and right side will generate the same compiled code

```
class FooInt
{
public:
     int& bar()
     {
         return subject;
     }
private:
     int subject;
}

class FooDouble
{
public:
     double& bar()
     {
         return subject;
     }
private:
     double subject;
}


FooInt fooInt;
FooDouble fooDouble;
```

inheritance Vs templates

inheritance

- **run time** polymorphism
- requires run time mechanism for **binding**
- **late** binding

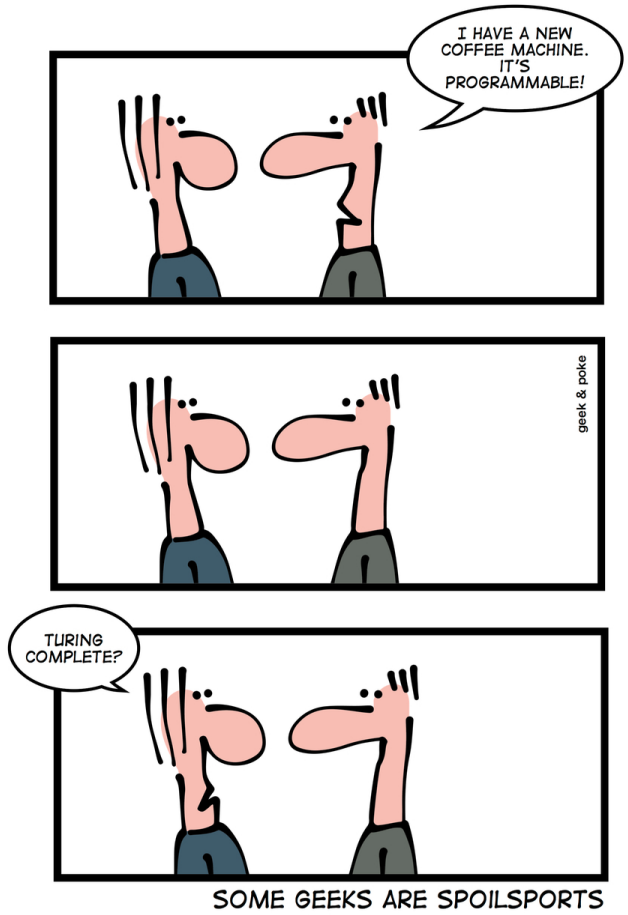
templates

- **compile time** polymorphism
- each set of different template parameters may cause the **generation** of a different internal function definition
- **no** run time **cost**

template → Turing-complete

- templates are a compile time mechanism in C++ that is Turing-complete
  - any computation expressible by a computer program can be computed, in some form, by a template metaprogram prior to runtime

- ***is this fact useful in practice?***



C++ templates with invalid parameters

compiler error messages
are often misleading and obscure

class Point

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

invalid type parameters

- the **properties** that a type parameter must satisfy are characterized **only implicitly** by the way instances of the type are used in the body of the template function

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

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

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

float mf2 = minValue(9.2,6);
// (float int) error: template argument deduction/substitution failed

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

Point p1(3.2,4.7);
Point p2(2.9,1.1);
Point p3 = minValue(p1,p2);
// error: no match for 'operator<' (operand types are 'Point' and 'Point')
```

Compiler error messages are often misleading and obscure

problems

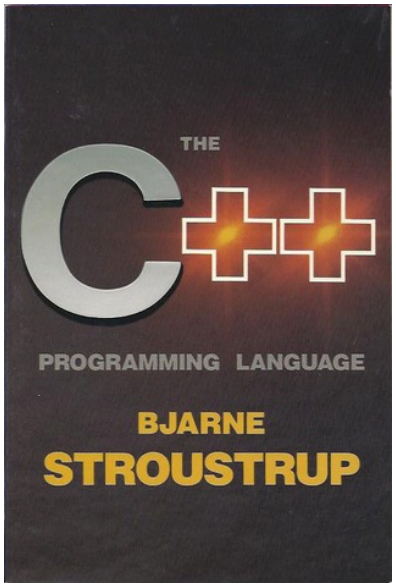
- the error message we get from `minValue(p1,p2)` is verbose and nowhere near as precise and helpful
- to use `minValue` we need to provide its definition, rather than just its declaration, this differs from ordinary code and changes the model of how we organize code
- the requirements of `minValue` on its argument type are implicit ("*hidden*") in its function body
- the error message for `minValue` will appear only when the template is instantiated, and that may be long after the point of call
- proposed solution:
  - using a **concept** we can get to the root of the problem by properly specifying a template's requirements on its arguments

C++ concepts

Concepts

Bjarne Stroustrup

The Future of Generic Programming and
how to design good concepts and use them well



once upon a time ...

*In about **1987**, I (Bjarne Stroustrup) tried to **design templates** with proper interfaces. I **failed**. I wanted three properties for templates:*

- ✔ *Full generality / expressiveness*
- ✔ *Zero overhead compared to hand coding*
- ✔ *Well-specified interfaces*

Then, nobody could figure out how to get all three, so we got

:) *Turing completeness*

:) *Better than hand-coding performance*

:(*Lousy interfaces (basically compile-time duck typing)*

*The lack of well-specified interfaces led to the **spectacularly bad error messages** we saw over the years. The other two properties made templates a **run-away success**.*

*The solution to the interface specification problem was named "**concepts**" by Alex Stepanov.*

`http://www.stroustrup.com/good_concepts.pdf`

duck typing

- *duck typing is an application of the duck test in type safety*
  - *“if it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck”*
- *type checking is deferred to runtime*
- *normal typing*
  - *suitability is assumed to be determined by an object's type only*
- *duck typing*
  - *suitability is determined by the presence of certain methods and properties (with appropriate meaning), rather than the actual type of the object*

timeline – traditional code

```
double sqrt(double d){...}; // C++84: accept any d that is a double

double d = 7;

double d2 = sqrt(d); // fine: d is a double

vector<string> vs = { "Good", "old", “templates" };

double d3 = sqrt(vs); // error: vs is not a double
```

- we have a function **`sqrt`** specified to require a double
- if we give it a double (as in **`sqrt(d)`**) all is well
- if we give it something that is not a double (as in **`sqrt(vs)`**) we promptly get a helpful error message, such as “a vector<string> is not a double.”

timeline – 1990s style generic code

```
template<class T> void sort(T& c) // C++98: accept a c of any type T
{
  // code for sorting (depending on various properties of T,
  // such as having [] and a value type with <
}
vector<string> vs = { "Good", "old", “templates" };
sort(vs); // fine: vs happens to have all the syntactic properties required by sort
double d = 7;
sort(d); // error: d doesn’t have a [] operator
```

- the **error message** we get from sort(d) is **verbose** and **nowhere** near as precise and **helpful**
  - will appear only when the template is instantiated, and that may be long after the point of call
- to use sort, **we need to provide its definition**, rather than just its declaration, this differs from ordinary code and changes the model of how we organize code
  - the requirements of sort on its argument type are implicit (“**hidden**”) in its function body

timeline – 2017(?)

```
// Generic code using a concept (Sortable):

void sort(Sortable& c); // Concepts: accept any c that is Sortable

vector<string> vs = { "Hello", "new", "World" };

sort(vs); // fine: vs is a Sortable container

double d = 7;

sort(d); // error: d is not Sortable (double does not provide [], etc.)
```

- this code is analogous to the sqrt example
- the only real difference is that
  - for double, a language designer (*Dennis Ritchie*) built it into the compiler as a specific type with its meaning specified in documentation
  - for Sortable, a *user* specified what it means in code
    - a type is Sortable if it has begin() and end() providing random access to a sequence with elements that can be compared using <
- we get an error message much as indicated in the comment
  - the message is generated immediately at the point where the compiler sees the erroneous call (sort(d))

concepts as constraints

- templates may be associated with a **constraint**, which specifies the **requirements** on template **arguments**, which can be used to select the most appropriate function overloads and template specializations
- constraints may also be used to **limit** automatic **type deduction** in variable declarations and function return types to only the types that **satisfy** specified requirements
- named sets of such requirements are called **concepts**
- each concept is a **predicate**, evaluated at **compile time**, and becomes a part of the interface of a template where it is used as a constraint
- violations of constraints are detected at **compile time**, early in the template instantiation process, which leads to **easy** to follow **error messages**

e.g. EqualityComparable

```
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;
}
```

- `Equality_comparable` is proposed as a **standard-library concept**
- like many concepts it takes more than one argument: ***concepts describe not just types but relationships among types***
- a `require` expression is never actually executed → the compiler looks at the requirements and compiles only if all are `true`

compiler errors

```
cout<<twoEquals(2,3,2)<<endl;           //(int int int) OK
cout<<twoEquals(9.2,6.1,5.8)<<endl;    //(float 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
```

shorthand notation

- standard notation example:

```
template <typename Seq>
     requires Sequence<Seq>
void algo(Seq& s) …
…
```

- shorthand notation:

```
template <Sequence Seq>
void algo(Seq& s) …
…
```

template<C T>

- means

```
template<typename T>
    requires C<T>
```

- we need the long form expressing complex requirements

concepts summary

- concepts are **named boolean predicates on template parameters**, evaluated at **compile time**
- a concept may be associated with a template (class template, function template, or member function of a class template); it serves as a **constraint** (**limits** the **set of arguments** that are **accepted** as template parameters)
- ***concepts = constraints + axioms***
- **constraints** are predicates on **static properties** of a type
- **axioms** state semantic requirements on types that should not be statically evaluated. An axiom is an *invariant* that is assumed to hold (as opposed required to be checked) for types that meet a concept
- **concepts** are predicates that represent general, abstract, and stable requirements of generic algorithms on their argument. They are defined in terms of constraints and axioms
- concepts **simplify compiler diagnostics** for failed template instantiations
  - if a programmer attempts to use a template argument that does not satisfy the requirements of the template, the compiler will generate an error. When concepts are not used, such errors are often difficult to understand because the error is not reported in the context of the call, but rather in an internal, often deeply nested, implementation context where the type was used

concept in 2017

- *during the C++11 standards development cycle, much work was done on a feature called "concepts" which aimed at providing systematic constraints on templates. Concepts was deferred from C++11 for lack of time to complete it, but work has continued*
- the non C++17 features (*think Concepts*) will be available as addons in upcoming compiler releases
- now we have
  - a ISO TS ("Technical Specification") for concepts
    - *"… This Technical Specification describes **extensions to the C++ Programming Language** that enable the specification and checking of **constraints on template arguments**, and the ability to overload functions and specialize class templates based on those constraints. These extensions include new syntactic forms and modifications to existing language semantics."*
  - concepts implementation is shipping as part of GCC 6
    - https://gcc.gnu.org/gcc-6/

ISO Technical Specification
*A Technical Specification addresses work **still under technical development**, or where it is believed that there will be a future, but not immediate, possibility of agreement on an International Standard. A Technical Specification is published for **immediate use**, but it also provides a means to obtain feedback. **The aim is that it will eventually be transformed and republished as an International Standard.***

GCC 6

- GCC 6 (GNU Compiler Collection) is the first compiler with concept support
- C++ Concepts are now supported when compiling with `--std=c++1z -fconcepts`
- `g++ --std=c++1z concepts.cpp -o concepts`
- `Online: http://en.cppreference.com/w/cpp/language/constraints`

GCC 6

- GCC 6 (GNU Compiler Collection) is the first compiler with concept support
- C++ Concepts are now supported when compiling with `--std=c++1z -fconcepts`
- `g++ --std=c++1z concepts.cpp -o concepts`
- `Online: http://en.cppreference.com/w/cpp/language/constraints`

concepts as semantic categories

- the intent of concepts is to **model semantic categories** rather than syntactic restrictions
- the ability to specify a **meaningful semantics** is a defining characteristic of a true concept, as opposed to a syntactic constraint
- *“Concepts are meant to express semantic notions, such as ‘a number’, ‘a range’ of elements, and ‘totally ordered.’ Simple constraints, such as ‘has a + operator’ and ‘has a > operator’ cannot be meaningfully specified in isolation and should be used only as building blocks for meaningful concepts, rather than in user code.”*

Avoid "concepts" without meaningful semantics

(ISO C++ core guideline T.20)

Bjarne Stroustrup - Herb Sutter

e.g. AdmitsSum

```
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);                  // OK
  cout<<mi<<endl;
  string s1 = mySum("alpha","beta");  // error?
  cout<<s1<<endl;
}
```

e.g. Number

```
template<typename T>
concept bool Number = requires(T a, T b) {
  { a+b } -> T;
  { a-b } -> T;
  { a*b } -> T;
  { a/b } -> T;
  { -a } -> T;
  { a+=b } -> T&;
  { a-=b } -> T&;
  { a*=b } -> T&;
  { a/=b } -> T&;
  { T{0} }; // can construct a T from a zero
  // …
} ;
```

this is extremely unlikely to be matched unintentionally

e.g. isIntegral

- Trait class that identifies whether T is an integral type.
- It inherits from integral_constant as being either true_type or false_type, depending on whether T is an integral type:

```
// is_integral example
#include <iostream>
#include <type_traits>

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;
  return 0;
}
```

- see also
  - is_floating_point
  - is_arithmetic
  - is_fundamental
  - is_signed
  - is_unsigned

concept overloading

- concepts allows us to select among functions based of properties of the arguments given
- Example: simple advance algorithm for iterator

```
void advance(Forward_iterator p, int n) {
   while(n--)
      ++p;
}
void advance(Random_access_iterator p, int n) {
   p+=n;
}
void use(vector<string>& vs, list<string>& ls) {
   auto pvs = find(vs,”foo”);
   advance(pvs,2);    // use fast advance
   auto pls = find(ls,”foo”);
   advance(pls,2);    // use slow advance
}
```

compiler overload resolution

- compiler **overload resolution** based on concepts:
  - if a function matches the requirements of one concept only, call it
  - if a function matches the requirements of no concept, the call is an error
  - if the function matches the requirements of two concepts, see if the requirements of one of those concepts is a subset of the requirements of the other
    - if so, call the function with the most requirements (the strictest requirements)
    - if not, the call is an error (ambiguous)

“A Concept Design for the STL”

- *Bjarne **Stroustrup**, Andrew **Sutton**, et al., “A Concept Design for the STL” (ISO/IEC JTC1/SC22/WG21 N3351, January **2012**)*
- *… We define seven concepts relating directly to language features: four that describe relationships between types and three that classify fundamental types. We imagine that these concepts will be intrinsic (i.e. , built into the compiler)*
- Type relations:
  - concept **Same**<typename T, typename U>
    - T and U denote exactly the same type after elimination of aliases
  - concept **Derived**<typename T, typename U>
    - T is derived from U
  - concept **Convertible**<typename T, typename U>
    - A value of T can be implicitly converted to a value of U
  - concept **Common**<typename T, typename U>
    - T and U can both be unambiguously converted to a third type C
- Type classification
  - concept **Integral**<typename T>
  - concept **SignedIntegral**<typename T>
  - concept **UnsignedIntegral**<typename T>