Chapter 16

Templates

Learning Objectives

- Function Templates
  - Syntax, defining
  - Compiler complications
- Class Templates
  - Syntax
  - Example: array template class
- Templates and Inheritance
  - Example: partially-filled array template class

Introduction

- C++ templates
  - Allow very "**general**" definitions for functions and classes
  - **Type names** are “parameters” instead of actual types
  - *Precise definition determined at run-time*

Function Templates

- Recall function swapValues:

```
1) void swapValues(int& var1, int& var2)
2) {
3)     int temp;
4)     temp = var1;
5)     var1 = var2;
6)     var2 = temp;
7) }
```

- Applies only to variables of **type int**
- But code would work for any types!

Function Templates vs. Overloading

- Could **overload** function for **chars**:

```
void swapValues(char& var1, char& var2)
{
    char temp;
    temp = var1;
    var1 = var2;
    var2 = temp;
}
```

- But notice: code is nearly identical!
  - Only difference is type used in 3 places

Function Template Syntax

- Allow "swap values" of **any type** variables:

```
1) template<class T>
2) void swapValues(T& var1, T& var2)
3) {
4)     T temp;
5)     temp = var1;
6)     var1 = var2;
7)     var2 = temp;
8) }
```

```
void swapValues(int& var1, int& var2)
{
        int temp;
        temp = var1;
        var1 = var2;
        var2 = temp;
}
```

- First line called "**template prefix**"
  - Tells compiler what's coming is "template"
  - And that T is a type parameter

Template Prefix

- Recall:
  template<class T>
- In this usage, "**class**" means "**type**", or "**classification**"
- Can be confused with other “known” use of word "class"!
  - C++ allows keyword "**typename**" in place of keyword "class" here
  - But most use "class" anyway

Template Prefix 2

- Again:
  template<class T>
- T can be **replaced** by any type
  - **Predefined** or **user-defined** (like a C++ class type)
- In function definition body:
  - T used like any other type
- Note: can use other than “T”, but T is "traditional" usage

Function Template Definition

- swapValues() function template is actually large "collection" of definitions!
  - A definition for each possible type!
- Compiler only generates definitions when required
  - But it's "as if" you'd defined for all types
- Write one definition → works for all types that might be needed

Calling a Function Template

- Consider following call:
  swapValues(int1, int2);
  - C++ compiler "**generates**" function definition for two int parameters using template
- Likewise for all other types
- Needn't do anything "special" in call
  - Required **definition automatically generated**

```
1)  template<class T>
    void swapValues(T& variable1, T& variable2)
2)  {
3)      T temp;
4)      temp = variable1;
5)      variable1 = variable2;
6)      variable2 = temp;
7)  }
8)  int main( ) {
9)      int integer1 = 1, integer2 = 2;
10)     cout << "Original integer values are " << integer1 << " " << integer2 << endl;
11)     swapValues(integer1, integer2);
12)     cout <<"Swapped integer values are "<< integer1<< " "<<integer2 << endl;

13)     char symbol1 = 'A', symbol2 = 'B';
14)     cout <<"Original character values are:"<<symbol1<< " "<<symbol2<< endl;
15)     swapValues(symbol1, symbol2);
16)     cout<<"Swapped character values are:"<<symbol1<<" "<<symbol2<< endl;
17)     return 0;
18) }
```

Another Function Template

- Declaration/prototype:

```
Template<class T>
void showStuff(int stuff1, T stuff2, T stuff3);
```

- Definition:

```
template<class T>
void showStuff(int stuff1, T stuff2, T stuff3)
{
    cout << stuff1 << endl
         << stuff2 << endl
         << stuff3 << endl;
}
```

showStuff Call

- Consider function call:
  showStuff(2, 3.3, 4.4);
- Compiler generates function definition
  - Replaces T with double
    - Since second parameter is type double
- Displays:
  2
  3.3
  4.4

```
template<class T>
void showStuff(int stuff1, T stuff2, T stuff3)
{
        cout << stuff1 << endl
             << stuff2 << endl
             << stuff3 << endl;
}
```

Compiler Complications

- Function declarations and **definitions**
  - Typically we have them separate
  - For templates → not supported on most compilers!
- **Safest** to place template *function definition in file where invoked*
  - Many compilers require it appear 1st
  - **Often we #include all template definitions**

More Compiler Complications

- Check your compiler's specific requirements
  - Some need to set special options
  - Some require special order of arrangement of template definitions vs. other file items
- **Most usable template program layout:**
  - Template definition in **same file** it's used
  - Ensure template definition **precedes** all uses
    - Can #include it

Examples for compiler compilation

```
// pfarray.h
1)  template<class T>
2)      class PFArray
3)      {
4)      public:
5)          PFArray(const PFArray<T>& pfaObject);
6)          void addElement(const T& element);
7)   …
8)          T& operator[](int index);
9)         PFArray<T>& operator =(const PFArray<T>& rightSide);
10) private:
11)         T *a; //for an array of T.
12) …
13)    };
```

Examples for compiler compilation

// **pfarray.cpp**

```
1)  template<class T>
2)  PFArray<T>::PFArray(const PFArray<T>& pfaObject):
        capacity(pfaObject.capacity), used(pfaObject.used)
3)  {…}

4)  template<class T>
5)  void PFArray<T>::addElement(const T& element)
6)  {
7)          …
8)          a[used] = element;
9)          used++;
10) }
```

Examples for compiler compilation

```
// Main.cpp
1)   #include "pfarray.h"
2)   #include "pfarray.cpp"
3)   …
4)   int main( )
5)   {
6)      PFArray<string> c(b);

7)      cout << "You copied the following:\n";
8)       count = c.getNumberUsed( );
9)       for (index = 0; index < count; index++)
                    cout << c[index] << " ";
10)      cout << endl;

11)      return 0;
12) }
```

Multiple Type Parameters

- Can have:
  template<class T1, class T2>
- Not typical
  - Usually only need one "replaceable" type
  - **Cannot** have "**unused**" template parameters
    - Each **must** be "**used**" in definition
    - Error otherwise!

Example taking two template parameters

```
1) template<class T1, class T2>
2) void PrintNumbers(const T1& t1Data, const T2& t2Data)
3) {
4)     cout << "First value:" << t1Data;
5)     cout << "Second value:" << t2Data;
6) }
```

```
1) PrintNumbers(10, 100);    // int, int
2) PrintNumbers(14, 14.5);   // int, double
3) PrintNumbers(59.66, 150); // double, int
```

Example- 3 or more type parameters

```
1)  template<class T1, class T2, class T3>
    T2 DoSomething(
                    const T1 tArray[],
                          T2 tDefaultValue,
                          T3& tResult    )
2)  {
3)      ...
4)  }
```

```
PrintNumbers(10, 100);    // int, int
PrintNumbers(14, 14.5);   // int, double
PrintNumbers(59.66, 150); // double, int
```

- Each call demands separate template instantiation
  - for the first and second types being passed (or say inferred).
- So, following three ***function template instances***
  - Template function (an "instance of a function template“)

  would be **populated by compiler**:

```
// const and reference removed for simplicity
void PrintNumbers(const int& t1Data, const int& t2Data);
void PrintNumbers(const int& t1Data, const double& t2Data);
void PrintNumbers(const double& t1Data, const int& t2Data);
```

Algorithm Abstraction

- Refers to implementing templates
- Express algorithms in "general" way:
  - Algorithm applies to variables of any type
  - Ignore incidental detail
  - Concentrate on substantive parts of algorithm
- Function templates are one way C++ supports **algorithm abstraction**

Defining Templates Strategies

- Develop function normally
  - Using actual data types
- Completely debug "ordinary" function
- Then convert to template
  - Replace type names with **type parameter** as needed
- Advantages:
  - Easier to solve "concrete" case
  - Deal with algorithm, not template syntax

Inappropriate Types in Templates

- Can use any type in template for which code makes "sense"
  - Code must behave in appropriate way
- e.g., swapValues() template function
  - Cannot use type for which assignment operator isn't defined
  - Example: an array:
    int a[10], b[10];
    swapValues(a, b);
    - Arrays cannot be "assigned"!

Selection Sort

```
1)/* a[0] to a[n-1] is the array to sort */
2)int i, j,  iMin;
3)for (j = 0; j < n-1; j++) {
4)    /* find the min element in the unsorted a[j .. n-1] */
5)    /* assume the min is the first element */
6)    iMin = j;
7)    /* test against elements after j to find the smallest */
8)    for ( i = j+1; i < n; i++) {
9)        /* if this element is less, then it is the new minimum */
10)       if (a[i] < a[iMin]) {
11)           /* found new minimum; remember its index */
12)           iMin = i;
13)       }
14)   }
15)
16)   if(iMin != j) {
17)       swap(a[j], a[iMin]);
18)   }
19)}
```

8
5
2
6
9
3
1
4
0
7

```
1)  template<class T>
2)  void sort(T a[], int numberUsed)
3)  {
4)      int indexOfNextSmallest;
5)      for (int index = 0; index < numberUsed - 1; index++)
6)      {//Place the correct value in a[index]:
7)          indexOfNextSmallest =  indexOfSmallest(a, index,
                                        numberUsed);
8)          swapValues(a[index], a[indexOfNextSmallest]);
9)        //a[0] <= a[1] <=...<= a[index] are the smallest of the
    original array
10)       //elements. The rest of the elements are in the remaining
    positions.
11)   }
12) }
```

```
1)   template<class T>
2)   int indexOfSmallest(const T a[], int startIndex, int numberUsed)
3)   {
4)       T min = a[startIndex];
5)       int indexOfMin = startIndex;

6)       for (int index = startIndex + 1; index < numberUsed; index++)
7)           if (a[index] < min)
8)           {
9)               min = a[index];
10)              indexOfMin = index;
11)              //min is the smallest of a[startIndex] through a[index]
12)          }

13)      return indexOfMin;
14)  }
```

Write your own function template!

Given

```
1)  int max(int nX, int nY)
2)  {
3)      return (nX > nY) ? nX : nY;
4)  }

1)  double max(double dX, double dY)
2)  {
3)      return (dX > dY) ? dX : dY;
4)  }
```

```
template <typename Type>
Type max(Type tX, Type tY)
{
    return (tX > tY) ? tX : tY;
}
```

```
1)int max(int nX, int nY)
2){
3)    return (nX > nY) ? nX : nY;
4)}
```

Explicit Instantiation

```
1)  template <class T>
    T GetMax (T a, T b)
2)  {
3)    T result;
4)    result = (a>b)? a : b;
5)    return (result);
6)  }

7)  int main () {
8)    int i=5, j=6, k;
9)    long l=10, m=5, n;
10)   k=GetMax<int>(i,j);
11)   n=GetMax<long>(l,m);
12)   cout << k << endl;
13)   cout << n << endl;
14)   return 0;
15) }
```

```
template <class T>
T GetMax (T a, T b)
{
  return (a>b?a:b);
}
int main ()
{
  int i=5, j=6, k;
  long l=10, m=5, n;
  k=GetMax(i,j);
  n=GetMax(l,m);
  cout << k << endl;
  cout << n << endl;
  return 0;
}
```

Class Template

Class Templates

- Can also "generalize" classes
  template<class T>
  - Can also apply to class definition
  - All instances of "**T**" in class definition replaced by **type** parameter
  - Just like for function templates!
- Once template defined, can declare objects of the class

Class Template Definition

```
template<class T>
class Pair
{
public:
    Pair();
    Pair(T firstVal, T secondVal);
    void setFirst(T newVal);
    void setSecond(T newVal);
    T getFirst() const;
    T getSecond() const;
private:
    T first;

    T second;
};
```

```
class Pair
{
public:
    Pair();
    Pair(int firstVal, int secondVal);
    void setFirst(int newVal);
    void setSecond(int newVal);
    int getFirst() const;
    int getSecond() const;
private:
    int first;

    int second;
};
```

Template Class Pair Members

- ```
  template<class T>
  Pair<T>::Pair(T firstVal, T secondVal)
  {
      first = firstVal;
      second = secondVal;
  }
  ```

Compare them!

- ```
  Pair::Pair(int firstVal, int secondVal)
  {
      first = firstVal;
      second = secondVal;
  }
  ```

Template Class Pair Members

- ```
  template<class T>
  Pair<T>::Pair(T firstVal, T secondVal)
  {
      first = firstVal;
      second = secondVal;
  }
  ```

- ```
  template<class T>
  void Pair<T>::setFirst(T newVal)
  {
      first = newVal;
  }
  ```

Pair Member Function Definitions

- Notice in member function definitions:
  - Each definition is itself a "template"
  - Requires template prefix before each definition
  - Class name before :: is "Pair<T>"
    - Not just "Pair"
  - But constructor name is just "Pair"
  - Destructor name is also just "~Pair"

Example- Member Templates

```
1)  struct Printer {
2)      std::ostream& os;
3)      Printer(std::ostream& os) : os(os) {}
4)      template<typename T>
        void operator()(const T& obj)   {    os << obj << ' ';   }
         // member template
5)  };
6)  int main()
7)  {
8)      std::vector<int> v= {1,2,3};
9)      std::for_each(v.begin(), v.end(), Printer(std::cout));
10)     std::string s = "abc";
11)     std::for_each(s.begin(), s.end(), Printer(std::cout));
12) …
13) }
```

Template Class Pair (noted difference: class template)

- Objects of class have "pair" of values of type T
- Can then declare objects:
  Pair<int> score;
  Pair<char> seats;
  - Objects then used like any other objects
- Example uses:
  score.setFirst(3);
  score.setSecond(0);

Class Templates as Parameters

- Consider:
  int addUP(const **Pair<int>**& the Pair);
  - The type (int) is supplied to be used for T in defining this class type parameter
  - It "happens" to be call-by-reference here
- Again: template types can be used anywhere standard types can

Class Templates Within Function Templates

- Rather than defining new overload:

```
template<class T>
    T addUp(const Pair<T>& the Pair);
//Precondition: Operator + is defined for values of type T
//Returns sum of two values in thePair
```

- Function now applies to all kinds of numbers

Restrictions on Type Parameter

- Only "reasonable" types can be substituted for T
- Consider:
  - **Assignment** operator must be "well-behaved"
  - **Copy constructor** must also work
  - If T involves pointers, then **destructor** must be suitable!
- Similar issues as function templates

Type Definitions

- Can define new "class type name"
  - To represent specialized class template name
- Example:
  ```
  typedef Pair<int> PairOfInt;
  ```
- Name "PairOfInt" now used to declare objects of type **Pair<int>**:
  ```
  PairOfInt pair1, pair2;
  ```
- Name can also be used as parameter, or anywhere else type name allowed

Friends and Templates

- Friend functions can be used with template classes
  - Same as with ordinary classes
  - Simply requires type parameter where appropriate
- Very common to have friends of template classes
  - Especially for operator overloads (as we've seen)

Pfarray.h

```
1)  template<class T>
2)  class PFArray   {
3)     public:
4)        PFArray( ); //Initializes with a capacity of 50.
5)        PFArray(int capacityValue);
6)        PFArray(const PFArray<T>& pfaObject);

7)        void addElement(const T& element);
8)        bool full( ) const; //Returns true if the array is full, false otherwise.
9)        int getCapacity( ) const;        int getNumberUsed( ) const;
10)       void emptyArray( );       //Resets used= 0, emptying the array.
11)       T& operator[](int index);   //access to elements 0 - numberUsed - 1.
12)       PFArray<T>& operator =(const PFArray<T>& rightSide);
13)       virtual ~PFArray( );
14)    private:
15)       T *a; //for an array of T.
16)       int capacity; //for the size of the array.
17)       int used; //for the number of array positions currently in use.
18)    };
```

Pfarray.cpp

```
1)   template<class T>
2)      PFArray<T>::PFArray( ) :capacity(50), used(0)
3)      {
4)         a = new T[capacity];
5)      }

6)   template<class T>
7)      PFArray<T>::PFArray(int size) :capacity(size), used(0)
8)      {
9)         a = new T[capacity];
10)     }


11)   template<class T>
12)     PFArray<T>::~PFArray( )
13)     {
14)        delete [] a;
15)     }
```

```
1) PFArrayD::PFArrayD( ) : capacity(50), used(0)
2) {
3)     a = new double[capacity];
4) }

5) PFArrayD::PFArrayD(int size):capacity(size), used(0)
6) {
7)     a = new double[capacity];
8) }
```

```
1)    template<class T>
2)      T& PFArray<T>::operator[](int index)    {
3)          if (index >= used)        {
4)              cout << "Illegal index in PFArray.\n";
5)              exit(0);
6)          }
7)         return a[index];
8)    }

9)    template<class T>
10)     PFArray<T>& PFArray<T>::operator =(const PFArray<T>& rightSide)
11)    {
12)        if (capacity != rightSide.capacity)       {
13)            delete [] a;
14)            a = new T[rightSide.capacity];
15)        }
16)        capacity = rightSide.capacity;
17)        used = rightSide.used;
18)        for (int i = 0; i < used; i++)    a[i] = rightSide.a[i];
19)        return *this;
20)    }
```

Modify to Template Version, plz!

```
1)PFArrayD::PFArrayD(const PFArrayD& pfaObject)
    :capacity(pfaObject.getCapacity( )),
    used(pfaObject.getNumberUsed( ))
1){
2)    a = new double[capacity];
3)    for (int i =0; i < used; i++)
4)        a[i] = pfaObject.a[i];
5)}
```

Copy constructor- template ver.

```
1)   template<class T>
2)     PFArray<T>::PFArray(const PFArray<T>& pfaObject)
3)       :capacity(pfaObject.getCapacity( )),
         used(pfaObject.getNumberUsed( ))
4)     {
5)       a = new T[capacity];
6)       for (int i =0; i < used; i++)        a[i] = pfaObject.a[i];
7)     }
```

Template Syntax

- **template** < parameter-list > **declaration**
- **declaration** of
  - a class (including struct and union),
  - a member class or member enumeration type,
  - a function or member function, a static data member at namespace scope, a variable or static data member at class scope, …
- **parameter-list-** a non-empty **comma-separated** list of the template parameters, each of which is **either**
  - **non-type** parameter, a **type** parameter, a **template** parameter, or a parameter **pack** of any of those.

Examples shown

Type template parameter

Type template parameter

typename *name*(optional)	(1)	
class *name*(optional)	(2)	
typename|class *name*(optional) **=** *default*	(3)	
typename|class ... *name*(optional)	(4)	(since C++11)

1) A type template parameter with an optional name
2) Exactly the same as 1)
3) A type template parameter with an optional name and a default
4) A type template parameter pack with an optional name

In the body of the template declaration, the name of a type parameter is a typedef-name which aliases the type supplied when the template is instantiated.

There is no difference between the keywords **class** and **typename** in a type template parameter declaration.

```
template<typename T> class X {}; // class template

struct A; // incomplete type
typedef struct {} B; // type alias to an unnamed type

int main()
{
    X<A> x1; // ok: 'A' names a type
    X<A*> x2; // ok: 'A*' names a type
    X<B> x3; // ok: 'B' names a type
}
```

Non-type template parameter

Non-type template parameter

type name(optional)	(1)	
type name(optional) = *default*	(2)	
type ... *name*(optional)	(3)	(since C++11)

1) A non-type template parameter with an optional name
2) A non-type template parameter with an optional name and a default value
3) A non-type template parameter pack with an optional name

type is one of the following types (optionally cv-qualified, the qualifiers are ignored)

- integral type
- enumeration
- pointer to object or to function
- lvalue reference to object or to function
- pointer to member object or to member function
- `std::nullptr_t` (since C++11)

Example Non-type template parameter

```
1)  template<const int* pci> struct X {};
2)  int ai[10];
3)  X<ai> xi;  // ok: array to pointer conversion and cv-qualification conversion

1)  struct Y {};
2)  template<const Y& b> struct Z {};
3)  Y y;
4)  Z<y> z;  // ok: no conversion and cv-qualification conversion

1)  template<int (&pa)[5]> struct W {};
2)  int b[5];
3)  W<b> w; // ok: no conversion

1)  void f(char);
2)  void f(int);
3)  template<void (*pf)(int)> struct A {};
4)  A<&f> a; // ok: overload resolution selects f(int)
```

Example II
Non-type template parameter

```
1)  template<int N> struct S { int a[N]; };
2)  // complicated non-type example
3)  template <    char c, // integral type
4)      int (&ra)[5], // lvalue reference to object (of array type)
5)      int (*pf)(int), // pointer to function
6)      int (S<10>::*a)[10] // pointer to member object (of type int[10])
7)  > struct Complicated {    // calls the function selected at compile time and
8)                            // stores the result in the array selected at compile time
9)      void foo(char base)     {       ra[4] = pf(c - base);    }
10) };
11) int a[5];
12) int f(int n) { return n; }
13) int main() {
14)     S<10> s; // s.a is an array of 10 int
15)     s.a[9] = 4;
16)     Complicated<'2', a, f, &S<10>::a> c;
17)     c.foo('0');
18)     std::cout << s.a[9] << a[4] << '\n';    // answer: 42
19) }
```

Template template parameter

Template template parameter

template < *parameter-list* > **typename**(C++17)|**class** *name*(optional)	(1)	
template < *parameter-list* > **typename**(C++17)|**class** *name*(optional) = *default*	(2)	
template < *parameter-list* > **typename**(C++17)|**class ...** *name*(optional)	(3)	(since C++11)

1) A template template parameter with an optional name

2) A template template parameter with an optional name and a default

3) A template template parameter pack with an optional name

Unlike type template parameter declaration, template template parameter declaration can only use the keyword **class** and not **typename**. (until C++17)

In the body of the template declaration, the name of this parameter is a template-name (and needs arguments to be instantiated)

```
template<typename T> class my_array {};
 // two type template parameters and one template template parameter:
template<typename K, typename V, template<typename> typename C = my_array>
class Map
{
    C<K> key;
    C<V> value;
};
```

Example of Template template parameter

```
1)  template<typename T> class A { int x; }; // primary template
2)  template<class T> class A<T*> { long x; }; // partial
    specialization
3)  // class template with a template template parameter V
4)  template<template<typename> class V> class C
5)  {
6)      V<int> y; // uses the primary template
7)      V<int*> z; // uses the partial specialization
8)  };
9)
10) C<A> c; // c.y.x has type int, c.z.x has type long
```

```
1)  // class template, with a type template parameter with a default
2)  template<typename T = float> struct B {};
3)   // template template parameter T has a parameter list:  one type template
    with a default
4)  template<template<typename = float> typename T> struct A
5)  {
6)      void f();
7)      void g();
8)  };
9)   // out-of-body member function template definitions
10) template<template<typename TT> class T>
11) void A<T>::f()
12) {
13)     T<> t; // error: TT has no default in scope
14) }
15) template<template<typename TT = char> class T>
16) void A<T>::g()
17) {
18)     T<> t; // ok: t is T<char>
19) }
```

Example II of Template template parameter

Predefined Template Classes

- Recall **vector** class
  - It's a template class!
- Another: basic_string template class
  - Deals with strings of "any-type" elements
  - e.g.,

basic_string<char> works for char's
basic_string<double> works for doubles
basic_string<YourClass> works for YourClass objects

std::basic_string

- Template < class CharT,
  class Traits = std::char_traits<CharT>,
  class Allocator = std::allocator<CharT>
  > class basic_string;
- Template parameters
  - CharT- character type
  - Traits- traits class specifying the operations on the character type
  - Allocator- Allocator type used to allocate internal storage

basic_string Template Class

- Already used it!
- Recall "string"
  - It's an alternate name for basic_string<char>
  - All member functions behave similarly for basic_string<T>
- basic_string defined in library <string>
  - Definition is in std namespace

Templates and Inheritance

- Nothing new here
- Derived template classes
  - Can derive from template or nontemplate class
  - Derived class is then naturally a template class
- Syntax same as ordinary class derived from ordinary class

1) template<typename **T**>
   class SmartItem : public Item<**T**>
2) {
3) };

- Class SmartItem is another class template which is inheriting from Item template
- If instantiates SmartItem with **char** type
  - Item<**char**> and SmartItem<**char**> would be **instantiated**!

Example of template-inheritance

```
1) template<size_t SIZE>
   class IntArray : public Array<int, SIZE>
2) {
3) };

4) int main()
5) {
6)     IntArray<20> Arr;
7)     Arr[0] = 10;
8) }
```

Templates and Virtual Functions

- Virtual Function
  - Employs late-binder
- Template
  - Employs early-binder, i.e. compile-time instantiation
- So,

```
1) class Sample
2) {
3) public:
4)     template<class T>
5)     virtual void Processor()  // ERROR!
6)     {    }
7) };
```

Why?

Templates and Virtual Functions

```
1) class Sample
2) {
3) public:
4)     template<class T>
5)     virtual void Processor()  // ERROR!
6)     {    }
7) };
```

Infinite specializations of Processor() can be overridden, depending on how method-template Processor is being called (via base of any of derived classes)
So, the virtual keyword loses its meaning

Summary

- Function templates
  - Define functions with parameter for a type
- Class templates
  - Define class with parameter for subparts of class
- Predefined vector and basic_string classes are template classes
- Can define template class derived from a template base class

std::for_each

http://en.cppreference.com/w/cpp/algorithm/for_each

Defined in header <algorithm>

```
template< class InputIt, class UnaryFunction >
UnaryFunction for_each( InputIt first, InputIt last, UnaryFunction f );                                   (1)

template< class ExecutionPolicy, class InputIt, class UnaryFunction2 >
void for_each( ExecutionPolicy&& policy, InputIt first, InputIt last, UnaryFunction2 f );                 (2)   (since C++17)
```

1) Applies the given function object f to the result of dereferencing every iterator in the range [first, last), in order.

2) Applies the given function object f to the result of dereferencing every iterator in the range [first, last) (not necessarily in order). The algorithm is executed according to policy. This overload does not participate in overload resolution unless `std::is_execution_policy_v<std::decay_t<ExecutionPolicy>>` is true.

For both overloads, if InputIt is a mutable iterator, f may modify the elements of the range through the dereferenced iterator. If f returns a result, the result is ignored.

Parameters

first, last - the range to apply the function to

policy - the execution policy to use. See execution policy for details.

f - function object, to be applied to the result of dereferencing every iterator in the range [first, last)

The signature of the function should be equivalent to the following:

```
void fun(const Type &a);
```

The signature does not need to have `const &`.
The type `Type` must be such that an object of type `InputIt` can be dereferenced and then implicitly converted to `Type`.

Type requirements

- InputIt must meet the requirements of InputIterator.
- UnaryFunction must meet the requirements of MoveConstructible. Does not have to be CopyConstructible
- UnaryFunction2 must meet the requirements of CopyConstructible.

Return value

1) f (until C++11) std::move(f) (since C++11)

2) (nothing)

Function objects

- A function object
  - Any object for which the function call operator is defined.
- Built-in function objects

Function objects

C++ defines several function objects that represent common arithmetic and logical operations:

Arithmetic operations	
`plus`	function object implementing `x + y` (class template)
`minus`	function object implementing `x - y` (class template)
`multiplies`	function object implementing `x * y` (class template)
`divides`	function object implementing `x / y` (class template)
`modulus`	function object implementing `x % y` (class template)
`negate`	function object implementing `-x` (class template)
Comparisons	
`equal_to`	function object implementing `x == y` (class template)
`not_equal_to`	function object implementing `x != y` (class template)

std::for_each

Possible implementation

```
template<class InputIt, class UnaryFunction>
UnaryFunction for_each(InputIt first, InputIt last, UnaryFunction f)
{
    for (; first != last; ++first) {
        f(*first);
    }
    return f;
}
```

```
1)   #include …
2)   struct Sum
3)   {
4)       Sum(): sum{0} { }
5)       void operator()(int n) { sum += n; }
6)       int sum;
7)   };
8)   int main()
9)   {
10)      std::vector<int> nums{3, 4, 2, 8, 15, 267};
11)      auto print = [](const int& n) { std::cout << " " << n; };
12)
13)      std::cout << "before:";
14)      std::for_each(nums.begin(), nums.end(), print);    std::cout << '\n';
15)      std::for_each(nums.begin(), nums.end(), [](int &n){ n++; });
16)       // calls Sum::operator() for each number
17)      Sum s = std::for_each(nums.begin(), nums.end(), Sum());
18)
19)      std::cout << "after: ";
20)      std::for_each(nums.begin(), nums.end(), print);    std::cout << '\n';
21)      std::cout << "sum: " << s.sum << '\n';
22)  }
```

Output:

```
before: 3 4 2 8 15 267
after:  4 5 3 9 16 268
sum: 305
```