Cairo University
Faculty of Computers and Information
Computer Science Department

Programming-2 CS213
2018/2019

Generic programming

Dr. Amin Allam

1 Generic programming in C++

Consider the following program which sorts an array:

```
1  template<class T>
2  void SelectionSort(T* a, int n)
3  {
4     int i,j;
5     for(i=0;i<n;i++)
6     {
7        // Keep the minimum of a[i ... j] in a[i]
8        for(j=i+1;j<n;j++)
9        {
10          if(a[j]<a[i])   // Less than operator of T
11          {
12                T temp=a[i];   // Copy constructor of T
13                a[i]=a[j];   // Assignment operator of T
14                a[j]=temp;   // Assignment operator of T
15          }
16       }
17       // Here: a[i] is the minimum of a[i ... n]
18       // Hence: a [0... i] are sorted
19    }
20 }
21
22 int main()
23 {
24    int a[]={3,5,2,1,7,4,6};
25    int n=sizeof(a)/sizeof(a[0]);
26    SelectionSort(a,n);
27    for(int i=0;i<n;i++) cout<<a[i]<<" "; cout<<endl;
28    return 0;
29 }
```

The `SelectionSort()` function is a template function which can sort an array whose elements are of type `T` such that type `T` contains the definition of copy constructor, copy assignment operator, and less than operator.

Consider the following program which utilizes the C++ standard template library:

```
#include <vector>
#include <algorithm>  // includes sort()
using namespace std;

int main()
{
   vector<int> v;    // creates a dynamic array object v
   v.push_back(4);   // adds the integer 4 to the end of v
   v.push_back(7);   // adds the integer 7 to the end of v
   v.push_back(2);

   // creates an iterator object iter which can traverse the elements of v
   // v.begin() is an iterator to the first element
   // v.end() is an iterator to the element after last
   // iterator allows similar processing of different containers (vector, set, ...)
   vector<int>::iterator iter;   // this loop works with any container:
   for(iter=v.begin(); iter!=v.end(); iter++) cout<<*iter;
   cout<<endl;   // prints: 4 7 2

   sort(v.begin(), v.end());   // sorts elements of v (works with any container)
   for(iter=v.begin(); iter!=v.end(); iter++) cout<<*iter;
   cout<<endl;   // prints: 2 4 7
   return 0;
}
```

Suppose we wish to implement similar behavior in C++ without using the standard libraries, we can proceed as the following:

```
template <class T>
class Vector
{
private:
   int n;
   T a[100];
public:
   Vector() {n=0;}
   void push_back(const T& x) {a[n]=x; n++;}

   typedef T* Iterator;

   Iterator begin() {return &a[0];}
   Iterator end() {return &a[n];}
};
```

```
1  template <class T, class Iterator>
2  void Sort(Iterator a, Iterator b)
3  {
4     for(Iterator i=a; i!=b; i++)
5     {
6        for(Iterator j=i+1; j!=b; j++)
7        {
8           if(*j<*i)
9           {
10             T temp=*i;
11             *i=*j;
12             *j=temp;
13          }
14       }
15    }
16 }
17
18 int main()
19 {
20    Vector<int> v;
21    v.push_back(4);
22    v.push_back(7);
23    v.push_back(2);
24
25    Vector<int>::Iterator iter;
26    for(iter=v.begin(); iter!=v.end(); iter++) cout<<*iter;
27    cout<<endl;
28
29    Sort<int, Vector<int>::Iterator>(v.begin(), v.end());
30    for(iter=v.begin(); iter!=v.end(); iter++) cout<<*iter;
31    cout<<endl;
32
33    return 0;
34 }
```

The only drawback of the previous program is that inside `Sort()`, we are not able to deduce the type `T` from the type `Iterator`. So, we needed to pass the type `T` to `Sort()` as shown in line 29. This issue is handled in the following version:

```
1  template <class T>
2  class IteratorBasedOnPointers
3  {
4  private:
5     T* i;
6  public:
7     typedef T value_type;
8     IteratorBasedOnPointers<T> () {}
9     IteratorBasedOnPointers<T> (T* const j) {i=j;}
10    void operator++(){i++;}
11    void operator++(int){i++;}
12    operator T*() {return i;}
13 };
14
15 template <class T>
16 class Vector
17 {
18 private:
19    int n;
20    T a[100];
21 public:
22    Vector() {n=0;}
23    void push_back(const T& x) {a[n]=x; n++;}
24
25    typedef IteratorBasedOnPointers<T> Iterator;
26
27    Iterator begin() {return &a[0];}
28    Iterator end() {return &a[n];}
29 };
30
31 template <class Iterator>
32 void Sort(Iterator a, Iterator b)
33 {
34    for(Iterator i=a; i!=b; i++)
35    {  for(Iterator j=i+1; j!=b; j++)
36       {  if(*j<*i)
37          {
38             typename Iterator::value_type temp=*i;
39             *i=*j; *j=temp;
40          }
41       }
42    }
43 }
```

We used the keyword `typename` to let the compiler know that `Iterator::value_type` is a type, not a variable. The compiler cannot conclude this fact when it compiles `Sort()` because the type of `Iterator` is not determined. Note that the compiler compiles a template function or class independently of any type, and also it compiles it once for each type substitution.

Suppose we wish to pass the comparison function as argument to the `Sort()` function instead of relying on the definition of the less than operator, we can overload the `Sort()` template function as follows:

```
template <class Iterator, class LessThan>
void Sort(Iterator a, Iterator b, LessThan IsLess)
{
   for(Iterator i=a; i!=b; i++)
   {
      for(Iterator j=i+1; j!=b; j++)
      {
         if(IsLess(*j, *i))
         {
            typename Iterator::value_type temp=*i;
            *i=*j;
            *j=temp;
         }
      }
   }
}
```

The previous definition of `Sort()` accepts a function which takes two variables of type `T` and returns `true` only if the first one is less than the second one. Additionally, it can accept a `function object` (also called `functor`) which is an object from a class which overloads the parentheses `()` operator to behave similarly to a function:

```
bool LessThan1(int a, int b) {return a<b;}

class LessThan2
{ public:
   bool operator()(int a, int b) {return a<b;}
};

int main()
{  // ...
   // Any one of the following lines works
   Sort(v.begin(), v.end());   // Use the less than operator
   Sort(v.begin(), v.end(), LessThan1);   // Use a global function
   Sort(v.begin(), v.end(), LessThan2());   // Use a functor (class object)
   // ...
}
```

2 Generic programming in C

Consider the following program which sorts an array using the standard `qsort()` function:

```
1  #include <stdlib.h>
2
3  int CompareInt(const void* pa, const void* pb)
4  {
5     int a=*(int*)(pa), b=*(int*)(pb);
6     return a-b;   // returns 0 if a==b, +ve if a>b, −ve if a<b
7  }
8
9  int CompareDouble(const void* pa, const void* pb)
10 {
11    double a=*(double*)(pa), b=*(double*)(pb);
12    double d=a-b;
13    // consider them equal if the difference is very small
14    if(d>-0.00001 && d<0.00001) return 0;
15    if(d>0) return 1;
16    return -1;   // d<0
17 }
18
19 int main()
20 {
21    int a[]={3,5,2,1,7,4,6};
22    int n=sizeof(a)/sizeof(a[0]);
23    qsort(a, n, sizeof(int), CompareInt);
24    for(int i=0;i<n;i++) cout<<a[i]<<" "; cout<<endl;
25
26    double b[]={3.1, 5.3, 2.2, 1.1, 7.4, 4.8, 6.9};
27    int m=sizeof(b)/sizeof(b[0]);
28    qsort(b, m, sizeof(double), CompareDouble);
29    for(int i=0;i<m;i++) cout<<b[i]<<" "; cout<<endl;
30
31    return 0;
32 }
```

The idea is to use the `void*` data type which accepts any pointer type. The first argument is a `void*` which accepts the array to be sorted. The second argument is the number of elements of the array. The third argument is the size of each element (because the function does not have any information about the type of elements). The fourth argument is a pointer to a function which compares two elements and returns 0 if they are equal, a positive value if the first is larger, and a negative value if the second is larger. The address of each element is passed as `void*` so match the function declaration and then a type case is performed from within the function according to the specific data type. The following is an implementation of a `Sort()` function with the same declaration of the standard `qsort()` function:

```
1  #include <stdlib.h>  // includes malloc()
2  #include <string.h>  // includes memcpy()
3
4  void Qsort(void* buf, int n, int elem_size,
5             int (*Compare)(const void*, const void*))
6  {
7     int i,j;
8     // allocate memory for the temporary variable
9     void* ptemp=malloc(elem_size);
10    for(i=0;i<n;i++)
11    {
12       for(j=i+1;j<n;j++)
13       {
14          void* pa=(char*)buf+(i*elem_size);   // pa=&a[i]
15          void* pb=(char*)buf+(j*elem_size);   // pb=&a[j]
16
17          if(Compare(pb, pa)<0)   // if(a[j]<a[i])
18          {
19             memcpy(ptemp, pa, elem_size);   // temp=a[i];
20             memcpy(pa, pb, elem_size);      // a[i]=a[j];
21             memcpy(pb, ptemp, elem_size);   // a[j]=temp;
22          }
23       }
24    }
25    // de-allocate memory of the temporary variable
26    free(ptemp);
27 }
28
29 int main()
30 {
31    int a[]={3,5,2,1,7,4,6};
32    int n=sizeof(a)/sizeof(a[0]);
33    Qsort(a, n, sizeof(int), CompareInt);
34    for(int i=0;i<n;i++) cout<<a[i]<<" "; cout<<endl;
35
36    double b[]={3.1, 5.3, 2.2, 1.1, 7.4, 4.8, 6.9};
37    int m=sizeof(b)/sizeof(b[0]);
38    Qsort(b, m, sizeof(double), CompareDouble);
39    for(int i=0;i<m;i++) cout<<b[i]<<" "; cout<<endl;
40
41    return 0;
42 }
```