Object-Oriented Programming: Template

Lectured by Ming-Te Chi 紀明德

First Semester, 2022

Computer Science Department
National Chengchi University

Slides credited from 李蔡彥 and 廖峻鋒

Template

- Why do we need template?
- Function template
  - Multiple template parameters
  - Templates and overloading
- Class template
  - Templates and classes
- Template Parameter
  - constant expression parameters
- Design considerations

The Problem of General Functions and Specific Data

- All you want to do is copy an array regardless of type. But the following won't work for arrays of doubles.

```
void Copy(int arrayTo[], int arrayFrom[], int n) {
    for(int i=0; i<n; i++)
        arrayTo[i]=arrayFrom[i];
}
int main() {
    int array1[]={1, 2, 3}, array2[3];
    Copy(array2, array1, 3);
    ...
}
```

Traditional C Solutions

```
// solution #1
#define MacroCopy(arrayTo, arrayFrom, n) {int i;\
for(i=0; i<n; i++) arrayTo[i]=arrayFrom[i]; }
```

Macro: ugly and error-prone

```
// solution #2
typedef int genericType;
void Copy(genericType arrayTo[], genericType arrayFrom[], int n){
    for(int i=0;i<n;i++)
        arrayTo[i] = arrayFrom[i];
}
```

Typedef: One type at a time

C++ Solution: Templates

- Template:

```
template<typename T>
void Copy(T arrayTo[], T arrayFrom[], int n) {
    for(int i=0; i < n; i++)
        arrayTo[i] = arrayFrom[i];
}
int main() {
    int iarray1[3] = {1, 2, 3}, iarray2[3];
    Copy(iarray2, iarray1, 3);
    double darray1[2] = {3.0, 4.0}, darray2[2];
    Copy(darray2, darray1, 2);
}
```

- Terms:
  - **Parameters:** the symbolic name that stands for the variable, e.g., T.
  - **Parameterized type:** a variable created through a template, e.g., `int`.
  - **Template instantiation:** what the compiler does when a template function is called.

note

- C++ 98
  - `template<class T>`

- `class seems be misleading, new name: typename`

- `Until C++ 11`
  - `template<typename T>`

- They are equivalent (except template template parameters in c++ 17).

Multiple Template Parameters

- For the above template, the following code is illegal.

```
int main() {
    int iarray[3]={1, 2, 3};
    double darray[3];
    Copy(darray, iarray, 3);
}
```

Error: function call 'Copy(double *, int *, int) does not match 'Copy(TYPETEMPLATE *, TYPETEMPLATE *, int)'

```
template<typename typeA, typename typeB>
void Copy(typeA arrayTo[], typeB arrayFrom[], int n){
    for(int i=0; i<n; i++) {
        arrayTo[i] = arrayFrom[i];
    }
}
```

Templates and Overloading

- You can overload templates.

```
template<typename type>
type Add(type x, type y) {
    return x+y;
}
template<typename type>
type Add(type x, type y, type z) {
    return x+y+z;
}
int main() {
    int x=5, y=4, z=1;
    cout << Add(x, y) << "\n";
    cout << Add(x, y, z) << "\n";
}
```

Output:
9
10

- The template below will work fine with integers, doubles, and chars, but not with c_strings or possibly other objects.

```
template<typename type>
bool GreaterThan(type x, type y){
    return x > y;
}
```

comparing pointers for c_strings?

Templates and Overloading Continued

- The solution to the previous problem is to provide an overloaded *non-template* function in addition to the template function.

```
template<typename type>
bool GreaterThan(type x, type y) {
    return x > y;
}
template<>
bool GreaterThan(char *string1, char *string2) {
    return strcmp(string1, string2)>0;
}
```

- Rule for "signature matching" with templates:

  Non-template functions have precedence over template functions in matching function calls.

Templates and Classes

- A general array class example:

```
template<typename type>
class ArrayT {
  public:
    ArrayT(int inSize);
    ~ArrayT();
    void insertElement(type element, int slot);
    type getElement(int slot) const;
  private:
    int arraySize;
    type *elements;
};
template<typename type>
ArrayT<type>::ArrayT(int inSize) {
    elements=new type[inSize]; // even here
    arraySize = inSize;
}
template<typename type>
ArrayT<type>::~ArrayT() {
    delete [] elements;
}
```

instantiate a class template

```
int main() {
    ArrayT<int> array(20);
    array.insertElement(10,0);
    cout << array.getElement(0);
}
```

Template Parameter

- **Types:** Types are the most often used template parameters.
- **Non-Types:**
  - lvalue reference
  - nullptr
  - pointer
  - enumerator
  - Integral Std::array<int, 3> myarray{1, 2, 3};

Templates With Constant Expression Parameters

- Templates can include *constant expression* in addition to *type parameters*.

```
template<typename type, int arraySize>
class ArrayT {
  public:
    void insertElement(type element, int slot);
    type getElement(int slot) const;
  private:
    type elements[arraySize];
};
template<typename type, int arraySize>
void ArrayT<type, arraySize>::insertElement(type e, int slot){
    if (slot<arraySize && slot <=0)
        elements[slot]=e;
    else
        cout<<"Warning, out of range!\n";
}
int main() {
    ArrayT<int, 100> array;
    array.insertElement(10, 99);
}
```

Design Considerations With Templates

- The main use for template is with *container* classes, i.e., Arrays, stacks, linked lists, etc.
- Avoid including elements to the template that will defeat its *generality*.
  - Example: add a function that adds together two components.

    Now you can't use the template on any class that doesn't overload +.
  - Example: you assume each object passed to the template contains a member function sort(), e.g.,

    Type.sort();
- Document the template thoroughly.
  - State which types will not work with the template.
  - State which functions you expect to be available.

Template problem in Separate Compilation (code)