

Template Functions I

- template functions are an easy way to generate many “similar” functions without writing code multiple times
- you write one template function and the compiler writes codes for multiple functions from your template
- in C you can't do this

Template Function II

- how can we write a template function to print an array of ints, doubles and strings
- in C you may need to implement the following three functions:

```
void  
printArrInt (int const *iarr, int const count);  
void  
printArrDouble (double const *darr, int const count);  
void  
printArrString (string const *sarr, int const count);
```

using %d, %f and %s respectively

Template Functions III

- what do you think of the following alternative?

```
template <typename TT> void  
printArr (TT const *arr, int const count);
```

- now you could use this as:

```
#define SMALL_ARR_LEN 5  
  
int iarr[SMALL_ARR_LEN] = { 1, 2, 3, 4, 5 };  
double darr[SMALL_ARR_LEN] = { 1.5, 2.4, 3.3, 4.2, 5.1 };  
char *sarr[SMALL_ARR_LEN] = { "Hello", "C++", "world,", "it", "rocks!" };  
int count = SMALL_ARR_LEN;  
  
printArr(iarr, count);  
printArr(darr, count);  
printArr(sarr, count);
```

Template Function IV

- implementation of the template function (goes in a .H file and **not** in a .C file):

```
template <typename TT> void
printArray (TT const *arr, int const count)
{
    assert(arr);
    assert(count >= 0);
    if (count == 1) {
        cout << arr[0] << endl;
    }
    else {
        int ii;
        for (ii = 0; ii < count - 1; ++ii)
            cout << arr[ii] << ", ";
        cout << arr[ii] << endl;
    }
}
```

Template Function V

- so where is the magic?
- from the call `printArr(iarr, count);`, the compiler figures out from the argument `iarr` that `TT` is `int`, i.e., `iarr` is an `int` array and hence it calls a function `printArr<int>(iarr, count);`
- then `cout << arr[ii]` “does the right thing” i.e. it prints out an `int`
- if the compiler can't figure out from the arguments the type of `TT`, you will have to specify it, e.g., for the following function:

```
template <typename TT> TT  
max (TT aa, TT bb);
```

- `max(1, 2);`, `max(1.2, 3.0);` are valid
- `max('a', 2);`, `max(1.2, 3);` are not!
- use `max<int>('a', 2);`, `max<double>(1.2, 3);` instead!

Template Class I

- defining a template class:

```
template <typename TT> class Stack {
private:
    TT *vals;
    int maxSize;
    int top;

public:
    Stack (int size);
    ~Stack (void);
    void push (TT val);
    TT pop (void);
};
```

Template Class II

- defining the member functions:

```
template <typename TT>
Stack<TT>::Stack (int size)
{
    assert(size > 0);
    maxSize = size;
    vals = new TT[size];
    top = 0;
}
template <typename TT>
Stack<TT>::~~Stack<TT> (void)
{
    delete [] vals;
}
```

Template Class III

- defining the member functions (cont.):

```
template <typename TT> void
Stack<TT>::push (TT val)
{
    if (top == maxSize) {
        cout << "Cannot push into the stack, stack full" << endl;
        exit(1);
    }
    else {
        vals[top] = val;
        ++top;
        cout << "Pushing element no. " << top << ": " << val
            << " into the stack" << endl;
    }
}
```

Template Class IV

- defining the member functions (cont.):

```
template <typename TT> TT
Stack<TT>::pop (void)
{
    if (top == 0) {
        cout << "Cannot pop from stack, stack empty" << endl;
        exit(1);
    }
    else {
        cout << "Popping element no. " << top << ": " << vals[--top]
            << " out of the stack" << endl;
    }
    return vals[top];
}
```

Template Class V

- using the template class:

```
int
main (int argc, char **argv)
{
    Stack<int> ist(N_SMALL);
    for (int ii = 0; ii < N_SMALL; ++ii)
        ist.push(2 * (ii + 1));
    for (int ii = 0; ii < N_SMALL; ++ii)
        (void)ist.pop( );
    cout << "=====" << endl;
    Stack<string> sst(N_SMALL);
    char tmp[128];
    // the following block should work fine
    for (int ii = 0; ii < N_SMALL; ++ii) {
        sprintf(tmp, "string %d", ii);
        sst.push(tmp);
    }
    for (int ii = 0; ii < N_SMALL; ++ii)
        (void)sst.pop( );
    return 0;
}
```

Template Class VI

- note, unlike template functions, the compiler never deduces the type of a class template object from its constructor, so you need to always need to be always explicit:

```
Stack<int> ist(N_SMALL);
```

- its possible to have more than one template parameters:

```
template <typename TypeOfFirst, typename TypeOfSecond> class DottedPair
{
    // stuff
};
```

- its possible to have default values for template parameters:

```
template <typename TypeOfFirst, typename TypeOfSecond = string> class DottedPair
{
    // stuff
};
```

or

```
template <typename TypeOfFirst = string, typename TypeOfSecond = string> class DottedPair
{
    // stuff
};
```

Code Files

```
prog2.H  
prog2.C  
prog2Makefile  
prog11.H  
prog11.C  
prog11Makefile
```