CSCI251/CSCI851 Spring-2021
Advanced Programming (**S6b**)

Generic Programming II:
Class templating

Outline

- Class templates and template classes.
- Non-type template parameters.
- Friends.
- Default types.
- Member function templates.
- Typeid on template classes.
- Variadic templates.

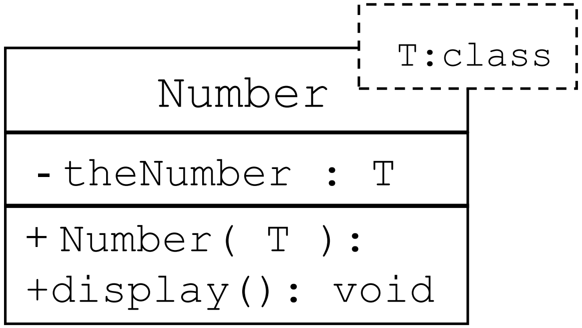
Warning: Error messages with templating can be horrendous.

Class templates and template classes

- In the previous set of notes we introduced templates for functions.
- If we need to create several similar classes, it may be useful to consider developing a **class template** in which at least one type is generic or parameterized.
- The syntax for class templating is similar to that for function templating.
- So we write a class template that the compiler turns into template classes.
- Classes are blueprints for objects, so class templates are blueprints for blueprints. ☺

- Here goes an example, with the UML …

```
template<typename T>
class Number {
    private:
        T theNumber;
    public:
        Number(const &T);
        void display();
};
```



- We need `Number<T>`, not just `Number`, to refer to the class.

```
template<typename T>
Number<T> :: Number(const &T n) {
    theNumber = n;
}

template<typename T>
void Number<T> :: display() {
    cout << theNumber << endl;
}
```

- We can declare and use some instances of this, with the type necessarily explicit in the instantiation.

```
int main() {

    Number<int> anInt(50);
    Number<double> aDouble(1.234);
    Number<char> aChar('A');

    anInt.display();
    aDouble.display();
    aChar.display();
//  anInt = aChar; //This won't work
}
```

Non-type Template parameters

- We saw this for function templates.

```
template<typename T, int ROWS, int COLS>
class Matrix
{
     . . .
  private:
     T data[ROWS][COLS];
};

Matrix<double, 3, 4> matr1;
Matrix<int, 2, 5> matr2;
…
```

Default types

- Here goes an example …
- … defaulting to an `int` with value `0`.

```
template <class T= int> class Numbers {
public:
    Numbers (T v = 0): val(v) { }
    …
private:
    T val;
};
```

Member function templates

- In class definitions, templated or not, member functions may have their own template parameters.

```
template<typename T>
class Storage {
      // ...
      public:
            template <typename R>
            void action(R first, R last);
      // ...
};
```

Typeid on template classes …

- The `typeid` operator can be applied to template classes.
- The type of an object that is an instance of a template class is in part determined by what data is used for its generic data when the object is instantiated.
- Two instances of the same template class that are created using different data are therefore considered to be different types.

```
template <typename T> class myclass {
  T a;
public:
  myclass(T i) { a = i; }
  // ...
};
int main()
{
  myclass<int> o1(10), o2(9);
  myclass<double> o3(7.2);

  cout << "Type of o1 is ";
  cout << typeid(o1).name() << endl;
  cout << "Type of o2 is ";
  cout << typeid(o2).name() << endl;
  cout << "Type of o3 is ";
  cout << typeid(o3).name() << endl;
  cout << endl;
  if(typeid(o1) == typeid(o2))
    cout << "o1 and o2 are of the same type\n";
  if(typeid(o1) == typeid(o3))
    cout << "Error\n";
  else
    cout << "o1 and o3 are different types\n";
  return 0;
}
```

Variadic templates

- A *variadic template* is a class or function template that supports an arbitrary number of arguments.
  - This mechanism is especially useful to C++ library developers because you can apply it to both class templates and function templates, and thereby provide a wide range of type-safe and non-trivial functionality and flexibility.

Variadic templates

- These are new to C++11, and are kind of cool.
- They allow an additional level of generalisation by replacing the keyword `typename` with `typename…` representing the use of a **parameter pack** of varying size.

```
template <typename T, typename… Args>
void fun(const T &t, const Args&… rest);
```

- `Args` is a **template parameter pack**, and `rest` a **function parameter pack**.

- Here goes an example of the usage … based on the example in the textbook.

```
int i=0, double d=3.14, string s= "red fish";

fun(i, s, 100, d);     // three pack parameters.
fun(s, 100, "Hello"); // two pack parameters.
fun(d, s);             // one pack parameter.
fun("Hello again");   // empty pack.
```

- How does the pack sizing work?
  - Remember that fun had a first parameter `const T &t`, and it's populated by the first argument.

The sizeof… operator.

- Yup, it's another ellipsis (…) and this is also new to C++11.

```
Template<typename … Args> void g(Args … args)
{
    cout << sizeof…(Args) << endl;
    cout << sizeof…(args) << endl;
}
```

- The two lines output, respectively, the number of type parameters and the number of function parameters.

A Variadic function template

- It should be apparent that dealing with variadic templates could be problematic, because we need to handle a variable number of types.
- To manage the variable number of types we often use recursive functions.
  - We will look at an example.

- Here we going to print one element off the pack at a time…
- Firstly the standard print …
- … then the variadic one.

```
template<typename T>
ostream &print(ostream &os, const T &t)
{
      return os << t;
}
```

```
template<typename T, typename… Args>
ostream &print(ostream &os, const T &t,
const Args&… rest)
{
      os << t << ", ";
      return print(os, rest…);
}
```

- The following does work on Banshee …

```
ostream &print(ostream &os)
{
        return os;
}

template<typename T, typename... Args>
ostream &print(ostream &os, const T &t, const Args&... rest)
{
        os << t << ", ";
        if (sizeof...(rest) > 0 )
                return print(os, rest...);
        else return os;
}

int main()
{
        print(cout, "a",3);
}
```

```cpp
#include <iostream>

using namespace std;

void print() {
    cout << endl;
}

template <typename T> void print(const T& t) {
    cout << t << endl;
}

template <typename First, typename... Rest> void print(const First& first, const Rest&... rest) {
    cout << first << ", ";
    print(rest...); // recursive call using pack expansion syntax
}

int main()
{
    print(); // calls first overload, outputting only a newline
    print(1); // calls second overload

    // these call the third overload, the variadic template,
    // which uses recursion as needed.
    print(10, 20);
    print(100, 200, 300);
    print("first", 2, "third", 3.14159);
}
```

Output

Copy

```
1
10, 20
100, 200, 300
first, 2, third, 3.14159
```