C++ Templates

and Java Generics

Generic Programming in C++

- Use **types** as **parameters** of functions and classes
- An additional level of **abstraction** when defining an algorithm, etc.
- **Generate** different code versions at **compilation** depending on **usage**
- So they cannot be in a separate “.cpp” file → must be **in a header file**
- **Warning:** parts not used in current code are not “checked” → **late errors**!

Before

"mymax" because "max" defined in std

```
int mymax(int a, int b) { return (a > b) ? a : b; }

double mymax(double a, double b) { return (a > b) ? a : b; }

char mymax(char a, char b) { return (a > b) ? a : b; }

int main() {
        cout << mymax(1, 3) << endl;
        cout << mymax(5.3, 3.2) << endl;
        cout << mymax('a', 'B') << endl;
}
```

After

keyword “class” equivalent to “typename” here

```
template <typename T>
T mymax(T a, T b) { return (a > b) ? a : b; }

int main() {
        cout << mymax<int>(1, 3) << endl;
        cout << mymax<double>(5.3, 3.2) << endl;
        cout << mymax<char>('a', 'B') << endl;
}
```

works with any type that has a “<” operator
even custom classes!

After

```
❯ g++ -S t2.cpp
❯ grep mymax t2.s
        call    _Z5mymaxIiET_S0_S0_
        call    _Z5mymaxIdET_S0_S0_
        call    _Z5mymaxIcET_S0_S0_
...
```

generate (text) assembly file

g++ generated 3 different versions

Template Classes

```
template <typename X, typename Y>
struct Pair {
        X first;
        Y second;
        Pair(X f, Y s) : first(f), second(s) {}
};

int main() {
        Pair<int, double> p1(1, 4.2);
        cout << p1.second << endl;
        Pair<char, char> p2('a', 'b');
        cout << p2.first << endl;
}
```

Dynamic Allocation

```
Pair<int, int> *p3 = new Pair<int, int>(3, 4);

cout << p3->first << endl;
```

Dealing with classes as type parameters

```
template <typename X, typename Y> struct Pair {
        X first;
        Y second;
        Pair(X f, Y s) : first(f), second(s) {}
};
struct A {
        A() { cout << "A" << endl; }
        A(const A& a) { cout << "copy A" << endl; }
};
int main() {
        A a1, a2;
        Pair<A, A> p(a1, a2);
}
```

prints:
A
A
copy A
copy A
copy A
copy A

Dealing with classes as type parameters

```
template <typename X, typename Y> struct Pair {
        X first;
        Y second;
        Pair(X& f, Y& s) : first(f), second(s) {}
};
struct A {
        A() { cout << "A" << endl; }
        A(const A& a) { cout << "copy A" << endl; }
};
int main() {
        A a1, a2;
        Pair<A, A> p(a1, a2);
}
```

prints:
A
A
copy A
copy A

But this doesn’t work then…

```
Pair<int, int> p2(1, 3);
```

```
❯ g++ pair2.cpp
pair2.cpp:18:20: error: cannot bind non-const lvalue reference of
type ‘int&’ to an rvalue of type ‘int’
   18 |   Pair<int, int> p2(1, 3);
```

needs…

```
int i = 1, j = 3;
Pair<int, int> p2(i, j);
```

Template Specialization

```
template <typename X, typename Y> struct Pair {
        X first;
        Y second;
        Pair(X& f, Y& s) : first(f), second(s) {}
};

template <> struct Pair<int, int> {
        int first;
        int second;
        Pair(int f, int s) : first(f), second(s) {}
};
```

Template Specialization

works ok…

```
A a1, a2;
Pair<A, A> p(a1, a2);
Pair<int, int> p2(1, 3);
```

Template Specialization -- Notes

- No need to specialize all type parameters

  e.g. `template <typename Y> Pair<int, Y>`

- All versions exists **simultaneously**!
- Each specialization can provide **completely different code**!
- But… we have to write versions for every primitive type

  (in the previous example)

Template Specialization -- Alternative

```
template <typename X, typename Y> struct Pair {
        X first;
        Y second;
        Pair(X f, Y s) : first(f), second(s) {}
};
```

specialize for pointers

```
template <typename X, typename Y> struct Pair<X*, Y*> {
        X* first;
        Y* second;
        Pair(X* f, Y* s) : first(f), second(s) {}
};
```

Template Specialization -- Alternative

```
int main() {
        A a1, a2;
        Pair<A*, A*> p(&a1, &a2);
        Pair<int, int> p2(1, 3);
        cout << p2.first << endl;
}
```

C++ Templates

Java Generics

Java Generics != C++ Templates

- Only **one version** of the code exists!
- Any reference to a type parameter is **replaced by Object**
- Known as “**type erasure**”
- Java will generate code without the need of **explicit** casts in every place
- Checks that different type values are not used for the same type parameter!

Java Generics

"<>" is the diamond operator
could have been
Pair<Integer, Integer>

```
public class A1 {
        public static void main(String[] args) {
                Pair<Integer, Integer> p1 = new Pair<>(1, 2);
                System.out.println(p1.first);
        }
}
class Pair<X, Y> {
        public X first;
        public Y second;
        Pair(X f, Y s) { this.first = f; this.second = s; }
}
```

Java Generics -- Similar to...

```java
public class A1Obj {
        public static void main(String[] args) {
                Pair p1 = new Pair((Integer) 1, (Integer) 2);
                System.out.println((Integer) p1.first);
        }
}
class Pair {
        public Object first;
        public Object second;
        Pair(Object f, Object s) { this.first = f; this.second = s; }
}
```

Java Generics -- Safe Types

```
Pair<Integer, Integer> p1 = new Pair<>(1, "a");
```

```
❯ javac A2.java
A2.java:3: error: incompatible types: cannot infer type arguments
for Pair<>
                Pair<Integer, Integer> p1 = new Pair<>(1, "a");
```

not allowed even if everything is replaced by Object