Generics

EECS 230

Winter 2017

(Monomorphic) max functions

```
int max(int x, int y)
{
        if (x < y) return y; else return x;
}
```

(Monomorphic) max functions

```
int max(int x, int y)
{
        if (x < y) return y; else return x;
}

double max(double x, double y)
{
        if (x < y) return y; else return x;
}
```

(Monomorphic) max functions

```
int max(int x, int y)
{
    if (x < y) return y; else return x;
}

double max(double x, double y)
{
    if (x < y) return y; else return x;
}

const string& max(const string& x, const string& y)
{
    if (x < y) return y; else return x;
}
```

A generic max function

```
template <typename T>
const T& max(const T& x, const T& y)
{
        if (x < y) return y; else return x;
}
```

A generic max function

```
template <typename T>
const T& max(const T& x, const T& y)
{
        if (x < y) return y; else return x;
}
```

This is actually std::max.

A (monomorphic) pair class

In Int_double_pair.h:

```
class Int_double_pair
{
public:
        Int_double_pair(int, double);

        int first() const;
        double second() const;

private:
        int first_;
        double second_;
};

bool operator==(const Int_double_pair&,
                const Int_double_pair&);
```

Int-double-pair implementation

In Int_double_pair.cpp

```
Int_double_pair::Int_double_pair(int first, double second)
            : first_(first), second_(second)
{ }

int Int_double_pair::first() const
{ return first_; }

double Int_double_pair::second() const
{ return second_; }

bool operator==(const Int_double_pair& a,
                const Int_double_pair& b)
{
    return a.first() == b.first() && a.second() == b.second();
}
```

What if we want a pair of a string and a char?

Well…

What if we want a pair of a string and a char?

Well…

```
class String_char_pair
{
public:
        String_char_pair(std::string, char);

        const std::string& first() const;
        char second() const;

private:
        std::string first_;
        char second_;
};

bool operator==(const String_char_pair&,
                        const String_char_pair&);
```

Introducing generics

A generic class is a class that works with multiple other types

Introducing generics

A generic class is a class that works with multiple other types

If we make a generic Pair class, then we can use it as:

- Pair<int, double>
- Pair<std::string, char>
- and many more!

Introducing generics

A generic class is a class that works with multiple other types

If we make a generic Pair class, then we can use it as:

- Pair<int, double>
- Pair<std::string, char>
- and many more!

We do this using a *template*

Interface for generic pair

In Pair.h:

```
template <typename T1, typename T2>
class Pair
{
public:
        Pair(const T1&, const T2&);
        const T1& first() const;
        const T2& second() const;

private:
        T1 first_;
        T2 second_;
};

template <typename T1, typename T2>
bool operator==(const Pair<T1, T2>&, const Pair<T1, T2>&);
```

Implementing templates

When we implement a class template:

- Every member function must templated as well.
- Templated definitions must be visible where they are used.
- This means that templated definitions usually *must* go in a header.

Implementation of generic pair

Also in Pair.h:

```
template <typename T1, typename T2>
Pair<T1, T2>::Pair(const T1& first, const T2& second)
            : first_(first), second_(second)
{ }

template <typename T1, typename T2>
const T1& Pair<T1, T2>::first() const
{ return first_; }

template <typename T1, typename T2>
const T2& Pair<T1, T2>::second() const
{ return second_; }

template <typename T1, typename T2>
bool operator==(const Pair<T1, T2>& a, const Pair<T1, T2>& b)
{ return a.first() == b.first() && a.second() == b.second(); }
```

Templates impose requirements on type parameters

For this to compile, operator== must be defined for T1 and T2, whatever they are:

```
template <typename T1, typename T2>
bool operator==(const Pair<T1, T2>& a, const Pair<T1, T2>& b)
{ return a.first() == b.first() && a.second() == b.second(); }
```

– To CLion! –