Generics

EECS 211

Winter 2018

(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 struct

In Int_double_pair.h:

```
struct Int_double_pair
{
      Int_double_pair(int, double);
      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 i, double d)
            : first(i), second(d)
{ }

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…

```
struct String_char_pair
{
    String_char_pair(std::string, char);
        std::string first;
        char second;
    }
};

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

Introducing generics

A generic class (or struct) is a class (or struct) that works with multiple other types

Introducing generics

A generic class (or struct) is a class (or struct) that works with multiple other types

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

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

Introducing generics

A generic class (or struct) is a class (or struct) that works with multiple other types

If we make a generic Pair struct, 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>
struct Pair
{
    Pair(const T1&, const T2&);
    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 or struct 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& v1, const T2& v2)
            : first(v1), second(v2)
{ }

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! –