CS11 Advanced C++

Winter 2020 – Class Templates

Generic Programs

- A lot of programs are independent of data type
  - Example: an **`abs()`** function
  - Example: your **`ConcurrentBoundedQueue`** class

- How to use a function or data structure with multiple different data types?

- One “solution” – make copies of the code, and edit the types…
  - A maintenance nightmare! *Never do this!!* ☺

C++ Templates

- C++ introduces templates
  - Allows a class or function to be parameterized on types and constants
- A template isn't itself a class or function…
  - More like a *pattern* for classes or functions
- To use a template, you instantiate it by supplying the template parameters
  - The result is a class or a function
  - "Generating a class/function from the template."

A Simple Template Example

- A **`Point`** class, which uses **`double`** values

```
class Point {
  double x_coord, y_coord;
public:
  Point() : x_coord{0}, y_coord{0} {}
  Point(double x, double y) : x_coord{x}, y_coord{y} {}
  double getX() const { return x_coord; }
  void setX(double x) { x_coord = x; }
  ...
};
```

- We want to have points with **`int`** coordinates, or **`float`** coordinates, or …
  - Let's make it a template!

The **`Point`** Class-Template

- A point class-template:

```
template<typename T> class Point {
  T x_coord, y_coord;
public:
  Point() : x_coord{0}, y_coord{0} {}
  Point(T x, T y) : x_coord{x}, y_coord{y} {}
  T getX() const { return x_coord; }
  void setX(T x) { x_coord = x; }
  ...
};
```

- Parameterized on coordinate-type, named **`T`**
  - Instead of **`double`**, just say **`T`** instead
  - "**`typename T`**" means any type – primitives or classes
  - Can use **`class`** instead of **`typename`** (means same thing)
  - "**`class`**" is a bit confusing since **`T`** can also be a primitive

Where Templates Live

- Templates generally live *entirely* in **`.h`** files
  - Unlike normal classes, *no* code in **`.cpp`** files
  - Code that uses a template must see the entire template definition
  - C++ compilers effectively treat them like big macros…
    - …with type-checking and many other capabilities.
- So, the **`Point`** template goes into **`point.h`**
  - *All* **`Point`** code goes into **`point.h`**
  - No more **`Point.cpp`**, now that it's a template

Using Our **`Point`** Template

- Using the template is just as easy:

```
Point<float> pF{3.2, 5.1}; // Float coordinates
```

  - "**`T`** means **`float`** everywhere in **`Point`** template"
  - The class' name is **`Point<float>`**
- Now we want a **`Point`** with integers

```
Point<int> pInt{15, 6};    // Integer coordinates
```

  - "**`T`** means **`int`** everywhere in **`Point`** template"
  - The class' name is **`Point<int>`**
- C++ makes a whole new class for each unique template instantiation

What Types Can **`Point`** Use?

```
template<typename T>
class Point {
  T x_coord, y_coord;
public:
  Point() : x_coord{0}, y_coord(0) {}
  Point(T x, T y) : x_coord(x), y_coord(y) {}
  T getX() const { return x_coord; }
  void setX(T x) { x_coord = x; }
  ...
};
```

- In *this* template, can only use certain types for **`T`**!
  - **`T`** must support initialization to 0
  - **`T`** must support copy-construction
  - **`T`** must support assignment

Enhancing the **`Point`** Template

- Now let's add a **`distanceTo()`** function.

```
template<typename T>
class Point {
  ...
  T distanceTo(const Point<T> &other) const {
    T dx = x_coord - other.getX();
    T dy = y_coord - other.getY();
    return (T) sqrt((double) (dx * dx + dy * dy));
  }
};
```

- Now, *what else* must **`T`** support??
  - Addition, subtraction, and multiplication!
  - Casting from **`T`** to **`double`**, and casting from **`double`** to **`T`**
  - The types we can use for **`T`** are pretty constrained now

Template Gotcha #1

- For many years, C++ had a major limitation:
  - You couldn't *explicitly* state the requirements of what operations `T` must support
  - If someone uses a template with a type that doesn't support required operations, you get many cryptic compiler errors
- C++20 introduces concepts
  - Allows these constraints on `T` to be stated explicitly
  - The compiler can give much more meaningful error messages when a bad type is used with a template
- When you write templates, clearly document what operations the template parameters must support
  - As support for C++20 grows, you can eventually add these specifications to your code as well as your comments!

Other Template Parameters

- Can parameterize templates on constant values

```
template<int size> class CircularQueue {
  char buf[size];  // Static allocation
  int head, tail;
public:
  CircularQueue() : head(0), tail(0) {}
  ...
};
```

  - **`size`** is a constant value, known at compile time
  - Can declare different size circular queues

```
CircularQueue<1024> bigbuf;
CircularQueue<16> tinybuf;
```

  - No dynamic memory management
    - Faster, smaller, easier to maintain

Multiple Template Parameters

- Can specify multiple parameters

```
template<typename T, int size>
class CircularQueue {
  T buffer[size];
  int head, tail;
  ...
};
```

- Parameters can also refer to previous parameters

```
template<typename T, T default>
class SparseVector {
  ...
  // Return value at index, or default value.
  T getElem(int index) {...}
};
```

  - Now 0 doesn't have to be the sparse-vector's default value

Large Functions in Templates

- Sometimes template-class functions are big
  - Can put the definition after template-class declaration
  - (Just like an inline member function declaration)
  - Must still declare like a template

```
template<typename T> class Point {
  ...
  T distanceTo(const Point<T> &other) const;
};

template<typename T> inline
T Point<T>::distanceTo(const Point<T> &other)
  const {
  ...
}
```