Class Templates

When “Type” gets complicated

Abstraction

- There are many situations where we may find ourselves writing similar code for multiple different situations within a program.
  - Consider the structure of an array – it's adaptable to work with any type, serving as a data structure that can store useful information of said type.

Abstraction

- In these situations, it may be helpful to write our code *abstractly* so that it can be reusable in numerous situations.
  - But how? Don't we need to know specific type information?

Abstraction

- The key to abstracting code is to determine the "least common denominator" of the code's functionality and requirements.
  - For our example of an array, note that it doesn't actually need to *do* anything with objects aside from storing and releasing them when requested.

Abstraction

- The key to abstracting code is to determine the "least common denominator" of the code's functionality and requirements.
  - Naturally, there are many situations where we do need some limited knowledge of the data to be input, even if it's not complete knowledge.
  - We will not work with this *quite* yet.

An Abstracted Object

- The usual solution for automatic resizing arrays includes a few additional ideas.
  - Often, we don't care about the *actual* placement of items within an array.
  - What matters? The order of the items… and sometimes, merely their presence.

An Abstracted Object

- The usual solution for automatic resizing arrays includes a few additional ideas.
  - If we give up having a constant index for an item, it can help us organize the array's contents and size more easily.
  - The idea: keep all contents as far to the left as possible, so we always know where all the empty spots are.

An Abstracted Object

- C++ has a built-in class that operates like this – the std::vector class.
  - We will model our new class after this.

`void*`

- To allow our `vector` class to store objects of any type, let's make use of the type `void*`.
- In C++, a pointer to any type can be cast to the type `void*`.
  - Note, however, that C++ has limited type tracking data.

Using `void*`

- As a result, a variable defined as follows can hold pointers to objects of any type.

```
void** genericArray = new void*[5];
```

An Abstracted Object

```
public class vector
{
   private:
   void** data;  //The data storage array.
   int count;    //Number of elements held.
   int capacity; //The array's actual size.

   public:
   vector();
   ~vector();
   void add(void* ref);
   void* &operator[](int i);
}
```

An Abstracted Object

```
public vector::vector()
{
    data = new void*[10];
    count = 0;
    capacity = 10;
}

public vector::~vector()
{
    delete[] data;
}
```

An Abstracted Object

```
void vector::add(void* obj)
{
    if(count == capacity)
    {
    void** temp = new void*[capacity * 2];
    for(int i=0; i<capacity; i++)
    temp[i] = data[i];

    delete[] data;
    capacity *= 2;
    data = temp;
    }

    data[count++] = obj;
}
```

An Abstracted Object

```
void* &vector::operator[](int index)
{
    if(index >= count || index < 0)
    {
        throw out_of_range();
    }

    return data[index];
}
```

An Abstracted Object

- We could write other methods for this class, but let's stop here.
  - Note that we can pass objects in and get objects out.
  - The `vector`'s user can cast the `void*` pointer back to its original type, assuming it knows that type.

An Abstracted Object

- Are there any potential problems with this object?

Class Templates

- One main problem that can be seen with our current `vector` object?
  - There's no way to restrict what types the `vector`'s user may enter.
  - What if the *user* of `vector` wants to restrict the `vector` to hold only a certain type of object?
  - The other detail – we're storing everything via pointer.

Class Templates

- There are other ways to handle abstraction aside from opening up access to all types at once.
  - Within Java, the functionality is known as *generic* programming.
  - In C++, it's managed differently – through "template" classes.

Class Templates

- A *generic* object is one which can take a type as a parameter.
  - This is useful to enforce *type safety*, whereby we can make sure that class inputs are limited to only appropriate types.
  - This "type parameter" then functions within the class definition (almost entirely) as if it were a real type.

Class Templates

- In C++, objects are not "generic" in the same sense as in Java.
  - Classes designed "generically" in C++ are **compiled** into separate type-specific versions at compile time.
  - One version is made for *every* combination of template parameters for that class.

Class Templates

- As generic class definitions in C++ are actually *templates*, they are compiled very differently.
  - The class's methods and such are not actually compiled until an instantiation is seen in code.
  - The type parameter on each instantiation is used to flesh out the template and fully define the corresponding class from the template.

Class Templates

- As generic class definitions in C++ are actually *templates*, they are compiled very differently.
  - Because of this, *all* member methods of a template class should be declared *and defined* within the header file.
  - The *template* definitions must be seen via `#include` in the code files which wish to use them with a specific type.

A First Class Template

```
template <typename T> class CustomVector
{
   private:
   T* data;      //The data storage array.
   int count;     //Number of elements held.
   int capacity; //Size of the array.

   public:
   static const int defaultSize = 10;
   //...
```

A First Class Template

```
//...

public:
    CustomVector();
    ~CustomVector();
    T add(T item);
    T& operator[](int i);

//...
```

A First Class Template

```
template <typename T>
public CustomVector<T>::CustomVector()
{
    data = new T[defaultSize];
    count = 0;
    capacity = defaultSize;
}

template <typename T>
public CustomVector<T>::~CustomVector()
{
    delete[] data;
}
```

A First Class Template

```
template <typename T> class CustomVector
{
    private:
    //...
```

Any names specified within the above angled brackets become something of a “type variable” – these are specified upon object instantiation, by users of the object we’re defining.

A First Class Template

```
template <typename T>
public CustomVector<T>::CustomVector()
{
    data = new T[defaultSize];
    c
    c
}

temp
publ
{
    delete[] data;
}
```

When we define the method in our usual manner, we need to redeclare the type parameter for the compiler to match it with the version in the class declaration.

A First Class Template

```
template <typename T> void CustomVector<T>::add(T item)
{
    if(count == capacity) {
    T* temp = new T[capacity * 2];
    for(int i=0; i<capacity; i++)
    temp[i] = data[i];

    delete[] data;
    capacity *= 2;
    data = temp;
    }

    data[count++] = item;
}
```

A First Class Template

```
template <typename T> void CustomVector<T>::add(T item)
{
    if(count == capacity) {
    T* temp = new T[capacity * 2];
    for(int i=0; i<capacity; i++)
    t
    d
    c
    d
    }
    d
}
```

This “type variable” is also used wherever the specified type constraints need to be utilized, as if the assigned name actually were a type.

Attempts in code to pass in an object of types not castable to T will cause *compiler* errors.

A First Class Template

```
template <typename T>
T &CustomVector<T>::operator[](int index)
{
    if(index >= count || index < 0)
    {
        throw out_of_range();
    }

    return data[index];
}
```

A First Class Template

```
template <typename T>
T &CustomVector<T>::operator[](int index)
{
    return data[index];
}
```

We can even use this “type parameter” as a *return* type, allowing us to guarantee users of our class that they will receive objects of the type they originally specified.

Method Templates

- Similar syntax may be used to create method templates that take type parameters.

```
template <typename T>
```

Summary

- The core motivation for the use of template programming is to promote *type safety* while having highly reusable methods or class definitions.
  - Rather than having to manually write separate versions of the "same" object for multiple types, templates do that work for us!