Object-Oriented Programming

CSCI-UA 0470-001
Class 12
Instructor: Randy Shepherd
Java Generics & C++ Templates
Parametric Polymorphism

• *Parametric polymorphism* is a way to make a language more expressive, while still maintaining *static type-safety*.

• A function or a class can be written such that it can handle *values* identically without depending on their *type*.

• Such functions and data types are called *generic* and form the basis of *generic programming*. 
Generic Programming

• *generic programming* is a style of programming in which algorithms are written in terms of *types to-be-specified-later*

• These types that are then instantiated when needed for specific types, provided as parameters

• Moreover, generics allow you to abstract over types.

• A common case is container types, such as a list or a set.
Those are words
Motivation

• We have a compiler for a reason. It finds bugs for us. Runtime bugs are generally much more difficult to solve.

• Remember our BetterPoint.java where we were able to use Enums to prevent runtime errors in our getCoordinate method?

  https://github.com/nyu-oop/point-java/blob/master/src/main/java/edu/nyu/oop/BetterPoint.java#L41
  vs
  https://github.com/nyu-oop/point-java/blob/master/src/main/java/edu/nyu/oop/Point.java#L36

• Similarly, generics give us another way of expressing some constraints to the compiler and therefore to prevent certain types of bugs.

• Generic programming also allows us in some cases to write less code, which is always good thing.
Java Generics

• Java Generics is an example of a generic programming language construct.

• Common examples are found in the Collections library.

• Have you seen something like this before?

    `List<String> list = new ArrayList<String>();`
Use in Java Collections

1. // Without Generics
2. ArrayList l1 = new ArrayList();
3. l1.add("hello");
4. String s1 = (String) l1.get(0);  // Note the cast

5. // With Generics
6. ArrayList<String> l2 = new ArrayList<String>();
7. l2.add("hello");
8. String s2 = l2.get(0);

• Note the cast on line 4

• The programmer knows what kind of data has been placed into the list.

• The compiler can only guarantee that an Object will be returned.

• To ensure the assignment to a variable of type String is type safe, the cast is required.
Use in Java Collections

There is the possibility of a runtime error, since the programmer may be mistaken.

Enter Generics. Notice the type declaration on line 7.

It specifies that this is a List of of String, written ArrayList<String>.

Why? Type-safety! The compiler can now catch certain errors.
Stack Implementation

- We can use Generics in our own data structures as well!
Lets look at some code...

https://github.com/nyu-oop/generics
Generic Implementation

• How are generics implemented by the Java compiler?
  
  • Type erasure -- removes generic type information from source code, adds casts as needed, and produces byte code
  
  • Essentially, the type parameters go away and the compiler generates exactly the same naive code we saw in our RawList.java
  
• Why is this the way the compiler works?
  
  • Backwards compatibility. this approach did not require the JVM to be changed
Generic Issues

• No generic arrays

• No instantiations of generic types
  • i.e. E e = new E(); where E is a type parameter.

• No primitive types, which require wrapper classes. int => Integer
  • There is “auto-boxing” to ease conversions from primitive types to boxed types and back again, but this has an overhead cost.

• Type information lost at runtime.
  • Can be very annoying, Especially when using reflection.
C++ Templates

• Templates are the C++ analog to Java Generics

• They are somewhat more flexible

• As with Java, templates can be made for classes and functions, and said templates allow classes and functions to intake many different types

• Different than Java, they work with primitives, you can instantiate type parameters and there are generic arrays.
C++ Templates

- As stated Generics are implemented by *erasure*

- C++ templates are implemented by *expansion*
  
  - Each instance of a template with a new type is generated and compiled
  
  - Moreover, if you have lists of strings, ints and doubles, you have 3 different functions generated.
  
  - This could lead to *code bloat*. In practice this is usually fine.
Template Example

• Class definition preceded by template keyword followed by type parameter.

• Similar syntax precedes implementation of functions.

• In both cases, type param name used in definition where concrete type would appear.

```cpp
template <class T>
class mypair {
    T a, b;
    public:
    mypair (T first, T second):
        a(first), b(second) {}
    T getmax ()
    { }
};

template <class T>
T mypair<T>::getmax ()
{
    T retval;
    retval = a>b? a : b;
    return retval;
}

int main () {
    mypair <int> intPair (100, 75);
    cout << intPair.getmax () << endl;
    mypair <string> stringPair ("a", "b");
    cout << stringPair.getmax () << endl;
    return 0;
}
Template Example

• At construction, the caller provides the type parameters after the typename and before the identifier.

• Any type that supports operations performed on it will work just fine.

• What does this code output?

```cpp
template <class T>
class mypair {
  T a, b;
public:
  mypair (T first, T second):
    a(first), b(second) {}
  T getmax () {
  }
};

template <class T>
T mypair<T>::getmax ()
{
  T retval;
  retval = a>b? a : b;
  return retval;
}

int main () {
  mypair <int> intPair (100, 75);
  cout << intPair.getmax () << endl;
  mypair <string> stringPair ("a", "b");
  cout << stringPair.getmax () << endl;
  return 0;
}
```
Template Example

• Key idea

• Data structures can be created and used to handle many different types without declaring separate classes for each.

• Anybody see an application for our translator?
Declaring and Defining Templates

- Class templates must be \emph{declared} in the header file.

- The \emph{compiler} creates the template for that type, replacing all instances of T with the actual type.

- Very different than Java!

- Class templates must also be \emph{defined} in the header file.
Template Specialization

• Specialized versions of class templates can be created for specific types

• Defined in .cpp (implementation) files because they don't need to be instantiated by the compiler

• Motivation, for a specific type we want to override the behavior of a certain method.

```cpp
template<class T>
class mycontainer {
    T element;
public:
    mycontainer(T arg) : element(arg) { }
    T increase() { return ++element; }
};

// class template specialization for char
template<>
class mycontainer<char> {
    char element;
public:
    mycontainer(char arg) : element(arg) { }
    char increase() {
        if ((element >= 'a') && (element <= 'z'))
            element += 'A' - 'a';
        return element;
    }
};

int main() {
    mycontainer<int> myint(7);
    mycontainer<char> mychar('j');
    cout << myint.increase() << endl;
    cout << mychar.increase() << endl;
    return 0;
}
```
Moar Code Plz

https://github.com/nyu-oop/templates
Templates in the Project

- We can use templates to make our translation of arrays much easier.
- Much better than implementing a array-class-per-type! (Especially since we cannot use inheritance).
- This is the only place we may use templates.
Templates in the Project

- Array templates along with a few specializations are implemented for you in java-lang-3
- However, *you* must copy it to your translator project.
- But before you do that, we will have an in-class exercise next week on this subject.
Arrays in java-lang-3

- Although you cannot see it here, we’ve moved our array into the __rt namespace.

- We have our forward declarations, those are templates as well.

- No typedefs, why?
Arrays in java-lang-3

- Our data layout looks basically the same, with the exception of utilizing the type parameter.

- Note that the constructor is defined inline. Since for template ‘implementation’ should go in the headers.

```cpp
template <typename T>
struct Array;

template <typename T>
struct Array_VT;

template <typename T>
struct Array {
    Array_VT<T>* __vptr;
    const int32_t length;
    T* __data;

    // The constructor (defined inline).
    Array(const int32_t length)
    : __vptr(&__vtable), length(length), __data(new T[length]()) {} 

    // The function returning the class object for the array.
    static java::lang::Class __class();

    // The vtable for the array.
    static Array_VT<T> __vtable;
};

// The vtable for arrays.
template <typename T>
Array_VT<T> Array<T>::__vtable;

// But where is the definition of __class()???
```
Arrays in java-lang-3

• As previously stated, all template ‘implementation’ code should go in header.

• So on line 26 we see our instantiation for the __vtable member.

• Where is the definition of __class()? Why is it not here?
Arrays in java-lang-3

- New vtable code, “now even more hard-to-readness!”
  (don’t you just love C++?)

- We have a typedef here to alias a pointer to an array of containing some type.

- Otherwise, really not much different than what we have seen in the past.
Arrays in java-lang-3

- Here is the specialization for an array of Objects.
- Basically just an implementation of the \texttt{\_\_class()} method.
- Why?
- Hmm.. that \texttt{\_\_Class} constructor look a little different, I think.
Arrays in java-lang-3

• Here is the specialization for an array of ints

• And now that I see this..there are definitely some changes to the Class constructor.

• Any guesses as to why?
Arrays in java-lang-3

- Note line 4
- Since we don’t have a class representing an int, we have to construct the class.

```java
// Template specialization for arrays of ints.

template<>
java::lang::Class Array<int32_t>::__class() {
    static java::lang::Class ik =
        new java::lang::Class(
            __rt::literal("int"),
            (java::lang::Class)__rt::null(),
            (java::lang::Class)__rt::null(),
            true
        );

    static java::lang::Class k =
        new java::lang::Class(
            literal("[I"),
            java::lang::__Object::__class(),
            ik);

    return k;
}
```
In-class exercise

• Next class we will do an in-class exercise to get you familiar with templates and the new arrays in java-lang

• In the meantime, work on your translator.

• The new java-lang version with templated arrays is on Github
  https://github.com/nyu-oop/java-lang-3