

Lecture 11: Generics

Introduction

A data structure is a way of organizing data to enable efficient usage. Ideally, a data structure should be flexible enough to work with different data types without requiring redundant code. Many programming languages, including C++, provide mechanisms to define *generic* (also known as *template*) functions and classes, which allow code reuse by working with multiple data types instead of defining separate versions for each type.

Generics

A generic function or class uses placeholder types that actual data types replace when the function is invoked or the class is instantiated. To declare a generic function or class, use the ‘**template**’ keyword, followed by a template parameter list:

```
template <template-parameter-list>
```

where each template parameter is in the form

```
typename identifier or class identifier
```

The keywords ‘**typename**’ and ‘**class**’ are interchangeable when defining templates.

Within the function or class body, the template identifier replaces actual data types. Below are examples of generic functions and generic classes:

Example:

```
template <typename T>
void Swap(T& a, T& b)
{
    T t = a;
    a = b;
    b = t;
}
```

```
template <class T>
class Item
{
    public:
        T value;
        int count;
};
```

```
template <typename T>
T Max(T data[],int n)
{
    T m = data[0];
    for(int i = 1;i < n;i += 1)
    {
        if(m < data[i])
        {
            m = data[i];
        }
    }
    return m;
}
```

```
template <class K, class V>
class Entity
{
    public:
        K key;
        V values[100];
        int size;
};
```

Key Observations:

- Not all members of a generic class need to be generic (see *Item* class).
- Multiple template parameters can be used (see *Entity* class).

Template Invocation & Instantiation

To invoke a generic function or instantiate a generic class object, the following syntaxes are used

• Function Invocation Syntax:

```
function-name [<data-type>] (argument-list)
```

• Class Instantiation Syntax:

```
class-name<data-type> identifier [(argument-list)];
```

Note: Specifying the data type is typically optional for functions, as the compiler can deduce it from arguments. However, explicitly providing type is mandatory for class instantiation.

Example:

Using the functions and classes from the previous example

```

int main()
{
    std::string words[] = {"apple", "orange", "banana", "peach", "grape"};
    Item<std::string> counter;
    Entity<std::string,int> group;
    Swap(words[1],words[3]); //after word[1] = peach, word[3] = orange
    cout << Max(words,5); // orange
    return 0;
}

```

Valid Template Type Substitutions

When invoking a generic function or instantiating a generic class, not all data types can be substituted for template parameters. A valid substitution must support all operations performed on the generic type within the function or class.

Example:

```

template <typename T>
T diff(const T& lhs, const T& rhs)
{
    return lhs - rhs;
}

template <typename T>
T same(const T& lhs, const T& rhs)
{
    return lhs + rhs;
}

int main()
{
    std::string a = "first", b = "second";
    std::string c = diff(a,b); //error: - not defined for std::string
    std::string d = same(a,b); //valid
    return 0;
}

```

Non-Type Template Parameter

A non-type template parameter allows the passing of constant values (such as integers, pointers, or references) to a template instead of just types. Specifically, it can be

- an integral type (includes bool and char).
- an enumeration type.
- a pointer or reference to a class object or member function, or a function/

Example:

```

template <class T,int size>
class Array
{
    T data[size];
public:
    T& operator[](int idx)
    {
        if(idx >= 0 && idx < size) {return data[i];}
        return std::out_of_range("out of bound");
    }
};

int main()
{
    Array<int, 5> myArray; // create an integer array of size 5
    myArray[0] = 42;
    std::cout << myArray[0] << "\n"; //output: 42
    return 0;
}

```