Data Structures & Programming

C++ Interlude
generic programming

Golnar Sheikhshab

Why generic functions?

```
void my_swap(int& A, int& B){
        int tmp = A;
        A = B;
        B = tmp;
}

void my_swap(string& A, string& B){
        string tmp = A;
        A = B;
        B = tmp;
}
```

```
void my_swap(float& A, float& B){
        float tmp = A;
        A = B;
        B = tmp;
}

void my_swap(patient& A, patient& B){
        patient tmp = A;
        A = B;
        B = tmp;
}
```

How to make a generic function?

Desired behavior: the function takes input of any type and work properly

```
// Definition:
template <typename T>
void my_swap(T& A, T& B){
        T tmp = A;
        A = B;
        B = tmp;
}
```

```
// Usage:
int main(){
        int A = 10, B = 20;
        my_swap(A, B); // A = 20, B = 10

        string strA = "Hello", strB = "Bye";
        my_swap(strA, strB); // A = "Bye", B = "Hello"
}
```

Test your understanding

You are given the following function. It works only for integer data type. Make it so it works with any data type (i.e. make it generic).

```
int Identity(int A){

    return A;

}
```

Non-generic ArrayLists for integer and string (.h file)

```
class intArrayList{
public:
        intArrayList(int capacity);
        bool empty() const;
        int count() const;
        void print() const;
        bool add(int new_entry);
        ~intArrayList();
protected:
        int* listArray_;
        int n_, capacity_;
};
```

```
class stringArrayList{
public:
        stringArrayList(int capacity);
        bool empty() const;
        int count() const;
        void print() const;
        bool add(string new_entry);
        ~stringArrayList();
protected:
        string* listArray_;
        int n_, capacity_;
};
```

Non-generic ArrayLists for integer and string (.cpp file)

```
intArrayList::intArrayList(int capacity){
        listArray_ = new int[capacity];
        n_=0;
        capacity_ = capacity;
}
bool intArrayList::empty() const{
        return n_==0;
}
int intArrayList::count() const{
        return n_;
}
intArrayList::~intArrayList(){
        delete [] listArray_;
}
```

```
stringArrayList::stringArrayList(int capacity){
        listArray_ = new string[capacity];
        n_=0;
        capacity_ = capacity;
}
bool stringArrayList::empty() const{
        return n_==0;
}
int stringArrayList::count() const{
        return n_;
}
stringArrayList::~stringArrayList(){
        delete [] listArray_;
}
```

Rest of .cpp files

```
void intArrayList::print() const{
	for (int i=0; i<n_; i++)
		std::cout << listArray_[i] << " ";
	std::cout << std::endl;
}

bool intArrayList::add(int new_entry){
	if (n_+1 > capacity_)
		return false;
	listArray_[n_] = new_entry;
	n_++;
	return true;
}
```

```
void stringArrayList::print() const{
	for (int i=0; i<n_; i++)
		std::cout << listArray_[i] << " ";
	std::cout << std::endl;
}

bool stringArrayList::add(string new_entry){
	if (n_+1 > capacity_)
		return false;
	listArray_[n_] = new_entry;
	n_++;
	return true;
}
```

Changing the code to accept float

1. Change the name of the class (it's a new class)

2. Change all the types from string to float

Making the class generic

1. Change the name of the class (it's a new class)
2. Change all the types from string to T
3. Immediately before the line with keyword class, add the following line:

    template<typename T>

Generic ArrayLists(.h file)

```
template <typename T>
class ArrayList{
public:
        ArrayList(int capacity){
                listArray_ = new T[capacity];
                n_=0;
                capacity_ = capacity;
        }
        bool empty() const{
                return n_==0;
        }
// the rest is in the next slide
```

Generic ArrayLists(.h file) (2)

```
int count() const{
        return n_;
}

void print() const{
        for (int i=0; i<n_; i++)
                std::cout << listArray_[i] << " ";
        std::cout << std::endl;
}

 ~ArrayList(){
        delete [] listArray_;
}
// the rest is in the next slide
```

Generic ArrayLists(.h file) (3)

```
        bool add(T new_entry){
                if (n_+1 > capacity_)
                        return false;
                listArray_[n_] = new_entry;
                n_++;
                return true;
        }
protected:
        T* listArray_;
        int n_, capacity_;
};
```

How to use a generic class?

```
#include <iostream>
#include "ArrayList.h"
using namespace std;
int main(){
	ArrayList<int> my_int_array_list(7);
	my_int_array_list.add(5);
	my_int_array_list.add(10);
	my_int_array_list.print();
	ArrayList<string> my_string_array_list(5);
	my_string_array_list.add("Hello");
	my_string_array_list.add("Bye");
	my_string_array_list.print();
}
```

Possible BaseTypes

How to decide if a generic function/class would work on a given base type?

If you make the function/class non-generic but specific to that base type would it work? If yes, the answer to the previous question is also yes. If no, the answer to the previous question is also no.

Example

what should class patient have so that the following code works?

```
template <typename T>
void my_swap(T& A, T& B){
        T tmp = A;
        A = B;
        B = tmp;
}
int main(){
       patient p1("name", 5), p2("name", 3);
       my_swap(p1, p2);
}
```