CS4414 Recitation 6

C++ templates

10/2024

Logistics

- HW 3 released on Canvas
- Due date:
  - Part 1. **10/11 (Friday)**
  - Part 2. **10/27 (Sunday)**
- **START EARLY**
  - This assignment takes more time than hw1 and hw2. Make sure to start early.
- Late submission
  - -5 points per day, maximum -15 (3 days late submission)

What is C++?

A federation of related languages, with four primary sublanguages

- **C**: C++ is based on C, while offering approaches superior to C. Blocks, statements, processor, built-in data types, arrays, pointers, etc., all come from C
- **Object-Oriented C++:** "C with Classes", classes including constructor, destructors, inheritance, virtual functions, etc.
- **Template C++:** generic programming language. Gives a template, define rules and pattern of computation, to be used across different classed.
- **STL(standard template library):** a special template library with conventions regarding containers, iterators, algorithms, and function objects

Overview

- C++ class inheritance
- C++ template

C++ Inheritance

C++ Hierarchical Inheritance

```
class BaseClass
{
// data members
// member functions
}
```

```
class DerivedClass1 : visibility_mode BaseClass
{
// data members
//  member functions
}
```

```
class DerivedClass2 : visibility_mode BaseClass
{
// data members
//  member functions
}
```

Recap: Access Specifiers

3 access specifiers for class variables and methods in C++:

- **public** - accessible outside the class
- **private** (default) - inaccessible outside the class
- **protected** - only accessible to inherited classes outside the class itself. More on Inheritance later…

Hierarchical Inheritance: Visibility Mode

Determines how base class features will be inherited by child

class DerivedClass1: **public** BaseClass
{// body}

Access specifiers of base class maintained as is (private remains private, public remains pub…)

class DerivedClass1: **private** BaseClass
{// body}

Public and protected access specifiers from base become private (i.e., inaccessible by derived class objects)

class DerivedClass1: **protected** BaseClass
{// body}

Public and protected access specifiers from base become private (i.e., inaccessible by derived class objects)

Exercise: Fill in the blanks

Base Class	Derived Class	Derived Class	Derived Class
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not inherited	Not inherited	Not inherited

Function Overloading

What happens if functions share the same name in the same scope?

- No problem! As long as…
  - At compile time, the compiler can can choose which overload to use based on types and number of arguments passed in by caller

https://learn.microsoft.com/en-us/cpp/cpp/function-overloading?view=msvc-170

Function Overloading

Function declaration element	**Used for overloading?**
Function return type	No
Number of arguments	Yes
Type of arguments	Yes
Presence or absence of ellipsis	Yes
Use of **typedef** names	No
Unspecified array bounds	No
const or **volatile**	Yes (when applied to entire function)
Reference qualifiers (**&** and **&&**)	Yes

What about function overloading with hierarchical inheritance?

```cpp
class BaseClass
{
public:
    int foo(int i)
    {
        std::cout << "foo(int): ";
        return i+1;
    }
};
```

```cpp
class DerivedClass : public BaseClass
{
public:
    double foo(double d)
    {
        std::cout << "foo(double): ";
        return d+1.1;
    }
};
```

```cpp
int main()
{
    DerivedClass dObject = DerivedClass();

    std::cout << dObject.foo(4) << std::endl;
    std::cout << dObject.foo(4.3) << std::endl;

    return 0;
}
```

Question: What will the program output?

A. foo(double): 5.1
foo(double): 5.4

B. foo(int): 5
foo(double): 5.4

C. Error

No overload resolution between class hierarchy in C++

C++ Template

THE BASIC IDEA IS EXTREMELY SIMPLE (LECTURE SLIDE)

As a concept, a template could not be easier to understand.

Suppose we have an array of objects of type int:

int myArray[10];

With a template, the user supplies a type by coding something like Things<long>.

Internally, the class might say something like:

```
template<Typename T>
T myArray[10];
```

THE BASIC IDEA IS EXTREMELY SIMPLE (LECTURE SLIDE)

As a concept, a template could not be easier to understand.

Suppose we have an array of objects of type int:

int myArray[10];

With a template, the user supplie ng>.

Internally, the class might say so

T behaves like a variable, but the “value” is some type, like in or myClass

template<**Typename T**>
T myArray[10];

YOU CAN ALSO TEMPLATE A CLASS (LECTURE SLIDE)

```
template<typename T>
class Things {
    T       myArray[10];
    T       getElement(int);
    void    setElement(int,T);
}
```

TEMPLATED FUNCTIONS (LECTURE SLIDE)

Templates can also be associated with individual functions. The entire class can have a type parameter, but a function can have its own (perhaps additional) type parameters

This really should require that T be a type supporting "comparable".

```
Template<typename T>
T max(T a, T b)
{
        return a>b? a : b;              // T must support a > b
}
```

Motivation

- Your boss wants you to build a digital calculator
- You come up with something like this

```
int subtract(int a, int b){
    return a-b;
}
int main(){
    int x = 10;
    int y = 7
    int x = subtract(x,y);
    std::cout << z << std::endl;
}
```

Motivation

- But calculators should be able to subtract floats and doubles too! And much more...
- So you come up with this...

```
int subtract(int a, int b){
    return a - b;
}
double subtract (double a, double b){
    return a - b;
}
float subtract(float a, float b){
    return a - b;
}

int main(){
    ……
}
```

Solution: function templates

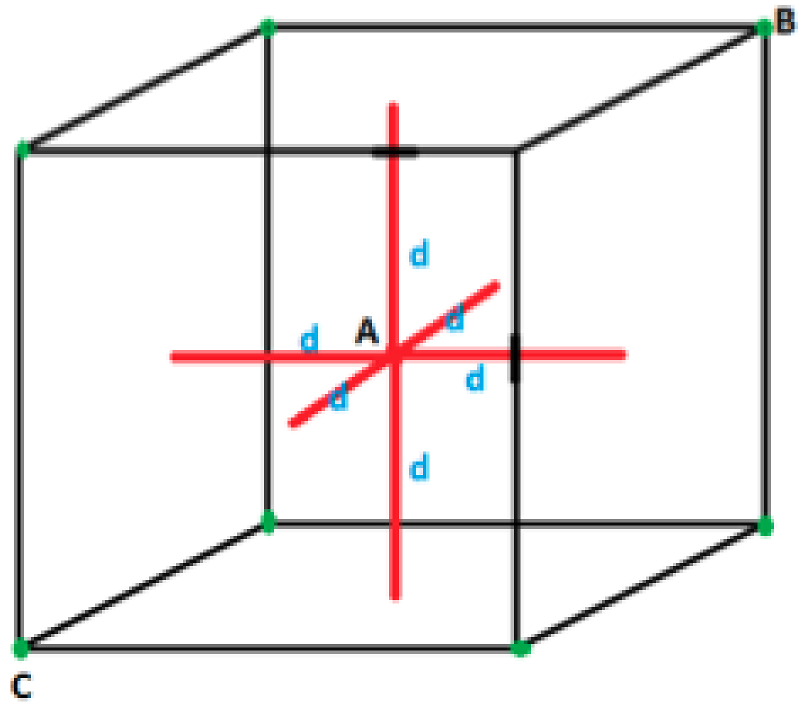
- What if you could just replace int with a generic data type
- How? Let's code!
- Limitation in shown example: parameters in subtract() must share type
- Uh-oh!
- No worries actually – let's code again!

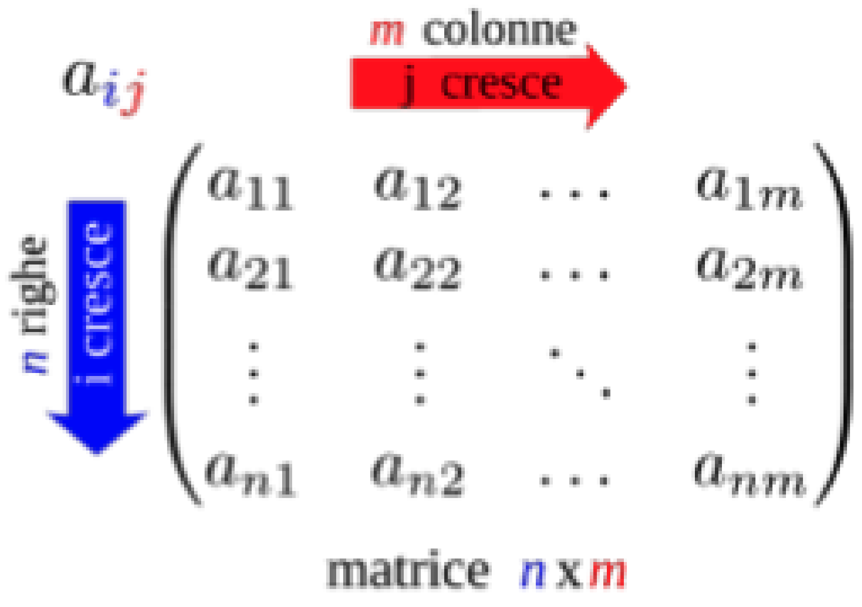
Metaprogramming with templates

- Metaprogramming: Metaprogramming is when a program takes as input another program. E.g., g++ takes your C++ program and transforms it into machine code
- With templates, we write a template for the actual source code

Templates enable generic programming

- A vector of objects, no matter what type, need similar memory management, indexing etc.





std::vector<T> can store	T=
A finite sequence of integers	int
A point in the n-dimensional space	double
A collection of Rectangles	Rectangle
Fields in a line of a csv file	std::string
A real matrix	std::vector<double>

Templates in the perspective of programming

- One larger goal of programming is to automate tasks. Reflexively, we want to avoid code copying in programming itself
- Functions are blocks of organized, reusable code that model a particular action
- Classes model similar set of objects
- Libraries provide a consistent set of features
- With templates, we can write functions or classes or variables that can work with different types. Templates abstract away the type.

Quick aside: template hpp files don't come with associated cpp files

- C++ often generates implementation file code internally for each type parameter from the template code in hpp file
- Remember: the compiler generates for each different type parameter that got used

How do we use templates when our function has an arbitrary number of parameters?

- Common issue…
  - Solution: Variadic templates
  - Let’s code

Variadic templates

```
class car {
public:
   int price;
   car(int price) : price(price) {}
};

class pc {
public:
   int price;
   pc(int price) : price(price) {}
};

class pen {
public:
   int price;
   pen(int price) : price(price) {}
};
```

Variadic templates

```
int sum() {
	return 0;
}
template <typename T, typename... Args>
int sum (T item, Args... rest) {
	return item.price + sum(rest...);
}

int main() {
	car c(100);
	pc pc(10);
	pen p(1);
	std::cout << "The sum is " << sum(c, pc, p);
}
```

Summary

- Templates let us move away from hardcoding types earlier on in our code so that our code can be more generic
- Templates allow us to specialize the treatment of select types while applying the default operations on all others
- C++ templates are compile-time constructs and thus must be implemented in a manner supporting such constraints
- Variadic templates let us leverage template benefits despite arbitrary number of parameters in function

Where to find the resources?

- Recitation references:
  - https://www.cs.cornell.edu/courses/cs4414/2021fa/Recitation_slides/CS4414%20Recitation%2010.pdf
  - https://www.cs.cornell.edu/courses/cs4414/2023sp/Recitation_slides/CS4414Recitation7Sp2023.pdf
- Effective C++: 55 specific ways to improve your programs and designs, Scott Meyers, 3rd edition
- A Tour of C++, Bjarne Stroustrup