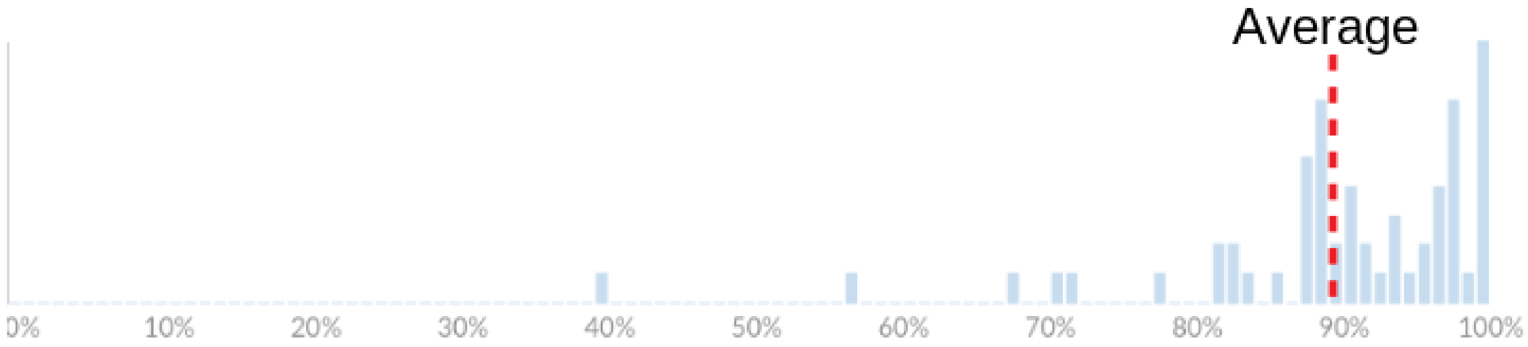
ECE 3574: Static Polymorphism using Templates

Changwoo Min

Readiness Exercise



- **Average: 8.1**
- **What does below average score mean?**
  - Lack of basic C++ knowledge
- **How to efficiently catch up C++ knowledge?**
  - Lynda.com C++ courses (free from VT library)

Milestone 0 Statistics (as of Jan 24)

Status	**%**
100% pass test cases	8%
Fail to pass test cases	8%
Compile error	64%
Didn't click the invitation link yet	17%
Didn't create a github ID yet	3%

Nightly Build & Test

- **What I will do**
  - Every midnight a buildbot will freshly clone your repository, compile and test it on the course reference environment.
  - It will send test results to your email.
- **What you should do**
  - Push your changes before midnight
  - It is okay to commit and push as many as you want/need
  - Ask questions to TA or me

Help Sessions

- Led by the TA
- If no one has shown up 10 min after the start time, the TA will leave
- **C++ review, course workflow, and lexing**
  - Mo 1/29 5:30-6:45pm, Surge 117A
  - Tu 1/30 6:30-7:45pm, WLH 330
- See Help session link in the course website

Meeting 4: Static Polymorphism using Templates

- Today we will look at how to reuse code using polymorphism and specifically static polymorphism through generic programming
  - Generics in C++ using Templates
  - Static Polymorphism
  - Exercise 04: How does `std::vector` work?

Generics in C++

- Templates elevate types to be generic, named but unspecified, and can work with functions and classes.
  - Template is roughly considered as a *type-checking macro*
- Templates allow code reuse as long as the types meet the functionality required by the template
- The C++ standard library uses templates extensively

Example 1: template function to swap

A simple example is a function to swap the contents of two variables (similar to `std::swap`):

```cpp
template< typename T >
void swap(T& a, T& b)
{
    T temp(b);
    b = a;
    a = temp;
}
```

The symbol `T` acts like a variable, in fact it is a type variable. Defined this way swap is generic, I can use it on any type that can be copied.

Example 1: template function to swap

```
int a = 1;
int b = 2;

std::cout << a << ", " << b << std::endl;
swap(a,b);
std::cout << a << ", " << b << std::endl;
```

```
// Template code
template< typename T >
void swap(T& a, T& b)
{
    T temp(b);
    b = a;
    a = temp;
}
```

```
// Compiler-generated code

void swap<int,int>(int& a, int& b)
{
    int temp(b);
    b = a;
    a = temp;
}
```

Example 1: template function to swap

```
std::string A = "foo";
std::string B = "bar";

std::cout << A << ", " << B << std::endl;
swap(A,B);
std::cout << A << ", " << B << std::endl;
```

```
// Template code
template< typename T >
void swap(T& a, T& b)
{
    T temp(b);
    b = a;
    a = temp;
}
```

```
// Compiler-generated code
void swap<std::string,std::string>(
          std::string& a, std::string& b)
{
    std::string& temp(b);
    b = a;
    a = temp;
}
```

Example 1: template function to swap

```
// If the type does not support a particular usage it generates a
// compile time error. For example suppose I wrote a class that
// explicitly does not allow copies

class NoCopy
{
public:
    // default constructor
    NoCopy() = default;

    // deleted copy constructor (i.e., llegal to use)
    NoCopy(const NoCopy & x) = delete;
};

// and tried to use swap as
NoCopy x,y;  // The default constructor will be called
swap(x,y);   // T temp(b) will try to call the deleted copy constructor

// My compiler complains
// swapexample.cpp:7:5: error: call to deleted constructor of T temp(b);
```

Example 2: template class to hold a pair of objects

Templates work with classes as well. For example, we might define a tuple holding two different types (aka std::pair) as

```
template <typename T1, typename T2>
class pair
{
public:
    pair(const T1& f, const T2& s);

    T1 first();
    T2 second();
private:
    const T1 m_first;
    const T2 m_second;
};
```

Example 2: template class to hold a pair of objects

And implement it like

```
template <typename T1, typename T2>
pair<T1,T2>::pair(const T1 & f, const T2 & s)
: m_first(f), m_second(s)
{}

template <typename T1, typename T2>
T1 pair<T1,T2>::first()
{
    return m_first;
}

template <typename T1, typename T2>
T2 pair<T1,T2>::second()
{
    return m_second;
}
```

Example 2: template class to hold a pair of objects

We might use it like so

```
pair<int,std::string> x(0, std::string("hi"));

std::cout << "First = " << x.first() << std::endl;
std::cout << "Second = " << x.second() << std::endl;
```

```
// Template code
template <typename T1, typename T2>
class pair
{
public:
    pair(const T1& f, const T2& s);

    T1 first();
    T2 second();
    // ...
};
```

```
// Compiler-generated code

class pair<int,std::string>
{
public:
    pair(const int& f, const std::s

    int first();
    std::string second();
    // ...
};
```

Organizing Template Code

The full implementation of a template must occur in the same translation unit (aka file). Thus they cannot be compiled and linked separately.

- We still would like to organize our code into a separate definition (header, .hpp) and implementation file (.cpp)
- Just include the implementation file at the bottom of the header file
- To prevent confusion the implementation file is often given a different extension (.tpp or .txx): Example

Exercise 04: How does `std::vector` work?

See Website

Useful C++ features

- Constructors and member initializer lists
- Operator `new` and `delete`
- Throwing an `exception`
- Copy Constructor in C++
- Copy constructor vs assignment operator in C++
- Copy constructors, assignment operators, and exception safe assignment
- Assignment Operators
- ECPP: 2. Constructors, Destructors, and Assignment Operators
- EMCPP: Item 7: Distinguish between () and {} when creating objects.

GDB: debugging on Linux

- gdb Cheatsheet

Next actions

- Read through a C++ standard library containers reference
- Reminder: Milestone 0 is due Monday 2/5.