CSE 374 - Week 8 (Fri)
C++: Templates, Smart Pointers

Instructor: Andrew Hu
TAs: Jim Limprasert, Kaelin Laundry, Keith Jiang, Nick Durand, Simon McFarlane, Yitian Hao
Plan for the Week

● **Monday: Intro to C++**
  ○ Operator overloading
  ○ References
  ○ Classes

● **Wednesday: RAII** *(Exercise released Wed, due Mon)*
  ○ RAII philosophy
  ○ Constructors & Destructors
  ○ new/delete

● **Friday: RAII in Practice** *(HW 6 released Fri, due Sat)*
  ○ Templates
  ○ STL, smart pointers
Suppose that...

- You want to write a function to compare two \texttt{ints}
- You want to write a function to compare two \texttt{strings}
  ○ Function overloading!

```cpp
// returns 0 if equal, 1 if value1 is bigger, -1 otherwise
int compare(const int& value1, const int& value2) {
    if (value1 < value2) return -1;
    if (value2 < value1) return 1;
    return 0;
}

// returns 0 if equal, 1 if value1 is bigger, -1 otherwise
int compare(const string& value1, const string& value2) {
    if (value1 < value2) return -1;
    if (value2 < value1) return 1;
    return 0;
}
```
Hm...

- The two implementations of `compare` are nearly identical!
  - What if we wanted a version of `compare` for every comparable type?
  - We could write (many) more functions, but that’s obviously wasteful and redundant

- What we’d prefer to do is write “generic code”
  - Code that is type-independent
  - Code that is compile-type polymorphic across types
C++ Parametric Polymorphism

- C++ has the notion of templates
  - A function or class that accepts a type as a parameter
    - You define the function or class once in a type-agnostic way
    - When you invoke the function or instantiate the class, you specify (one or more) types or values as arguments to it
  - At compile-time, the compiler will generate the “specialized” code from your template using the types you provided
    - Your template definition is NOT runnable code
    - Code is only generated if you use your template
Function Templates

- **Template to compare two “things”:**

```cpp
#include <iostream>
#include <string>

// returns 0 if equal, 1 if value1 is bigger, -1 otherwise
template <typename T> // <...> can also be written <class T>
int compare(const T &value1, const T &value2) {
    if (value1 < value2) return -1;
    if (value2 < value1) return 1;
    return 0;
}

int main(int argc, char **argv) {
    std::string h("hello"), w("world");
    std::cout << compare<int>(10, 20) << std::endl;
    std::cout << compare<std::string>(h, w) << std::endl;
    std::cout << compare<double>(50.5, 50.6) << std::endl;
    return EXIT_SUCCESS;
}
```
Compiler Inference

- Same thing, but letting the compiler infer the types:

```cpp
#include <iostream>
#include <string>

// returns 0 if equal, 1 if value1 is bigger, -1 otherwise
template <typename T>
t int compare (const T &value1, const T &value2) {
    if (value1 < value2) return -1;
    if (value2 < value1) return 1;
    return 0;
}

int main(int argc, char **argv) {
    std::string h("hello"), w("world");
    std::cout << compare(10, 20) << std::endl;  // ok
    std::cout << compare(h, w) << std::endl;   // ok
    std::cout << compare("Hello", "World") << std::endl;  // hm...
    return EXIT_SUCCESS;
```
Template Non-types

- You can use non-types (constant values) in a template:

```cpp
#include <iostream>
#include <string>

// return pointer to new N-element heap array filled with val
// (not entirely realistic, but shows what’s possible)
template <typename T, int N>
T* valarray(const T &val) {
    T* a = new T[N];
    for (int i = 0; i < N; ++i)
        a[i] = val;
    return a;
}

int main(int argc, char **argv) {
    int *ip = valarray<int, 10>(17);  // 1
    string *sp = valarray<string, 17>("hello");  // 2
    ...
}
```
What’s Going On?

- The compiler doesn’t generate any code when it sees the template function
  - It doesn’t know what code to generate yet, since it doesn’t know what types are involved

- When the compiler sees the function being used, then it understands what types are involved
  - It generates the *instantiation* of the template and compiles it (kind of like macro expansion)
    - The compiler generates template instantiations for *each* type used as a template parameter
Class Templates

- Templates are useful for classes as well
  - (In fact, that was one of the main motivations for templates!)

- Imagine we want a class that holds a pair of things that we can:
  - Set the value of the first thing
  - Set the value of the second thing
  - Get the value of the first thing
  - Get the value of the second thing
  - Swap the values of the things
  - Print the pair of things
#ifndef PAIR_H_
define PAIR_H_

template <typename Thing> class Pair {

public:
  Pair() { };

    Thing get_first() const { return first_; } 
    Thing get_second() const { return second_; } 
void set_first(Thing &copyme);
void set_second(Thing &copyme);
void Swap();

private:
    Thing first_, second_; 

};

#include "Pair.cc"

#endif   // PAIR_H_
Pair Function Definitions

template<typename Thing>
void Pair<Thing>::set_first(Thing &copyme) {
    first_ = copyme;
}

template<typename Thing>
void Pair<Thing>::set_second(Thing &copyme) {
    second_ = copyme;
}

template<typename Thing>
void Pair<Thing>::Swap() {
    Thing tmp = first_;
    first_ = second_
    second_ = tmp;
}

template<typename T>
std::ostream &operator<<(std::ostream &out, const Pair<T>& p) {
    return out << "Pair(" << p.get_first() << ", " "
    " << p.get_second() << ")";
}
#include <iostream>
#include <string>
#include "Pair.h"

int main(int argc, char** argv) {
    Pair<std::string> ps;
    std::string x("foo"), y("bar");

    ps.set_first(x);
    ps.set_second(y);
    ps.Swap();
    std::cout << ps << std::endl;

    return EXIT_SUCCESS;
}
Class Template Notes

- **Thing** is replaced with template argument when class is instantiated
  - The class template parameter name is in scope of the template class definition and can be freely used there
  - Class template member functions are template functions with template parameters that match those of the class template
    - These member functions must be defined as template function outside of the class template definition (if not written inline)
      - The template parameter name does *not* need to match that used in the template class definition, but really should
  - Only template methods that are actually called in your program are instantiated (but this is an implementation detail)
Review: new/delete
new/delete

- To allocate on the heap using C++, you use the `new` keyword instead of `malloc()` from `stdlib.h`
  - You can use `new` to allocate an object (e.g. `new Point`)
  - You can use `new` to allocate a primitive type (e.g. `new int`)

- To deallocate a heap-allocated object or primitive, use the `delete` keyword instead of `free()` from `stdlib.h`
  - Don’t mix and match!
    - Never `free()` something allocated with `new`
    - Never `delete` something allocated with `malloc()`
    - Careful if you’re using a legacy C code library or module in C++
Poll Question: PollEv.com/andrewhu
Poll Question ([PollEv.com/andrewhu](PollEv.com/andrewhu))

This code has a memory error. How should we fix it?

A. Add "delete ptr2"
B. Add "delete ref3"
C. Remove "delete ptr1"
D. Move "delete ptr1" to the end

```cpp
void print_int_ptr(int* ptr2) {
    cout << *ptr2 << endl;
    // delete ptr2;
}

void print_int_ref(int& ref3) {
    cout << ref3 << endl;
    // delete ptr3;
}

int main() {
    int* ptr1 = new int;
    *ptr1 = 42;
    print_int_ptr(ptr1);
    delete ptr1;
    print_int_ref(*ptr1);
}
```
C++ Smart Pointers

- Wouldn't it be nice if pointers just got `delete`'d for us?
- A **smart pointer** is an *object* that stores a pointer to a heap-allocated object
  - A smart pointer looks and behaves like a regular C++ pointer
    - By overloading `*`, `->`, `[ ]`, etc.
  - These can help you manage memory
    - The smart pointer will delete the pointed-to object *at the right time* including invoking the object’s destructor
      - When that is depends on what kind of smart pointer you use
    - With correct use of smart pointers, you no longer have to remember when to `delete` new’d memory!
A Toy Smart Pointer

- We can implement a simple one with:
  - A constructor that accepts a pointer
  - A destructor that frees the pointer
  - Overloaded \* and -> operators that access the pointer
ToyPtr Class Template

```cpp
#ifndef TOYPTR_H_
define TOYPTR_H_

template <typename T> class ToyPtr {
    public:
        ToyPtr(T* ptr) : ptr_(ptr) { }       // constructor
        ~ToyPtr() { delete ptr_; }         // destructor

        T& operator*() { return *ptr_; }    // * operator
        T* operator->() { return ptr_; }    // -> operator

    private:
        T* ptr_;                          // the pointer itself
    };
#endif // TOYPTR_H_
```

ToyPtr<MyClass> tp();

tp->field;

MyClass copy = *tp;
ToyPtr Example

```cpp
#include <iostream>
#include "ToyPtr.h"

// simply struct to use
typedef struct { int x = 1, y = 2; } Point;
std::ostream &operator<<(std::ostream &out, const Point &rhs) {
  return out << "(" << rhs.x << "," << rhs.y << ")";
}

int main(int argc, char **argv) {
  // Create a dumb pointer
  Point *leak = new Point;

  // Create a "smart" pointer (OK, it's still pretty dumb)
  ToyPtr<Point> notleak(new Point);

  std::cout << "  *leak: " << *leak << std::endl;
  std::cout << "  leak->x: " << leak->x << std::endl;
  std::cout << "  *notleak: " << *notleak << std::endl;
  std::cout << "  notleak->x: " << notleak->x << std::endl;

  return EXIT_SUCCESS;
}
```

What Makes This a Toy?

● Can’t handle:
  ○ Arrays
  ○ Copying (!)
  ○ Reassignment
  ○ Comparison
  ○ … plus many other subtleties…
Copying ToyPtr

- What happens when we try to copy it?

```c
void take_toy_obj(ToyPtr<int> z) {
    // Do nothing, let z be destructed
}

int main(int argc, char** argv) {
    ToyPtr<int> x(new int(5));
    ToyPtr<int> y = x;
    take_toy_obj(x);  // Calls cctor
    return EXIT_SUCCESS;
}
```
Demo: ToyPtr
Ownership of the Pointer

- Problem: ToyPtr assumes it has ownership of the pointer
  - Ownership means there is only one copy of this pointer
  - When ToyPtr gets destructed, it deletes it even if there are others using that pointer

- Solution: Reference counting is a technique for managing resources by counting and storing the number of references (i.e. pointers that hold the address) to an object
std::shared_ptr

- `shared_ptr` is similar to `ToyPtr` but we allow shared objects to have multiple owners
  - The copy/assign operators *increment* reference counts as needed
    - After a copy/assign, the two `shared_ptr` objects point to the same pointed-to object and the (shared) reference count is 2
  - When a `shared_ptr` is destroyed, the reference count is *decremented*
    - When the reference count hits 0, we `delete` the pointed-to object!
shared_ptr Example

#include <cstdlib>
#include <memory>

void take_shared_obj(std::shared_ptr<int> z) {
    // refcount:
    // Do nothing, let z be destructed
} // Calls destructor

int main(int argc, char** argv) {
    std::shared_ptr<int> x(new int(5)); // refcount: 1
    std::shared_ptr<int> y = x; // refcount: 2
    take_shared_obj(x); // Calls cctor
    return EXIT_SUCCESS; // refcount: 2
} // Calls destructor

use_shared.cc
Summary

- A `shared_ptr` allows shared objects to have multiple owners by doing *reference counting*
  - `delete` an object once its reference count reaches zero

- You can replace every normal pointer usage with a `shared_ptr`
  - No more memory leaks!
  - Technically, there are edge cases, but out of the scope of CSE 374

```cpp
C++ 14   make_shared<pair>(42, 374);
```
C++’s Standard Library

- C++’s Standard Library consists of four major pieces:
  1) The entire C standard library
  2) C++’s input/output stream library
     - `std::cin`, `std::cout`, `stringstreams`, `fstreams`, etc.
  3) C++’s standard template library (STL)
     - Containers, iterators, algorithms (sort, find, etc.), numerics
  4) C++’s miscellaneous library
     - Strings, exceptions, memory allocation, localization
STL Containers 😊

- A **container** is an object that stores (in memory) a collection of other objects (elements)
  - Implemented as class templates, so hugely flexible

- Several different classes of container
  - **Sequence containers** (vector, deque, list, ...)
  - **Associative containers** (set, map, multiset, multimap, bitset, ...)
  - Differ in algorithmic cost and supported operations
STL Containers 😞

- STL containers store by *value*, not by *reference*
  - When you insert an object, the container makes a *copy*
  - If the container needs to rearrange objects, it makes copies
    - *e.g.* if you sort a `vector`, it will make many, many copies
    - *e.g.* if you insert into a `map`, that may trigger several copies
  - What if you don’t want this (disabled copy constructor or copying is expensive)?
    - Use smart pointers!
Our Tracer Class

- **Wrapper class for an** `unsigned int value_`
  - Also holds unique `unsigned int id_` (increasing from 0)
  - Default ctor, cctor, dtor, `op=, op<` defined
  - `friend function operator<<` defined
  - Private helper method `PrintID()` to return "(id_, value_)" as a string
  - Class and member definitions can be found in `Tracer.h` and `Tracer.cc`

- **Useful for tracing behaviors of containers**
  - All methods print identifying messages
  - Unique `id_` allows you to follow individual instances
STL `vector`

- A generic, dynamically resizable array
  - Elements are stored in *contiguous* memory locations
    - Elements can be accessed using pointer arithmetic if you’d like
    - Random access is $O(1)$ time
  - Adding/removing from the end is cheap (amortized constant time)
  - Inserting/deleting from the middle or start is expensive (linear time)
vector/Tracer Example

```cpp
#include <iostream>
#include <vector>
#include "Tracer.h"

using namespace std;

int main(int argc, char** argv) {
    Tracer a, b, c;
    vector<Tracer> vec;

    cout << "vec.push_back " << a << endl;
    vec.push_back(a);
    cout << "vec.push_back " << b << endl;
    vec.push_back(b);
    cout << "vec.push_back " << c << endl;
    vec.push_back(c);

    cout << "vec[0]" << endl << vec[0] << endl;

    return EXIT_SUCCESS;
}
```
shared_ptr and STL Containers

- Safe to store shared_ptr in containers, since copy/assign maintain a shared reference count

```cpp
vector<std::shared_ptr<int> > vec;

vec.push_back(std::shared_ptr<int>(new int(9)));
vec.push_back(std::shared_ptr<int>(new int(5)));
vec.push_back(std::shared_ptr<int>(new int(7)));

int &z = *vec[1];
std::cout << "z is: " << z << std::endl;

std::shared_ptr<int> copied = vec[1]; // works!
std::cout << "*copied: " << *copied << std::endl;

std::cout << "vec[1].get(): " << vec[1].get() << std::endl; // no leaks!
```
STL iterator

- Each container class has an associated iterator class (e.g. `vector<int>::iterator`) used to iterate through elements of the container
  - **Iterator range** is from `begin` up to `end` i.e., `[begin, end)`
    - `end` is one past the last container element!
  - Some container iterators support more operations than others
    - All can be incremented `++`, copied, copy-constructed
    - Some can be dereferenced on RHS (e.g. `x = *it;`)
    - Some can be dereferenced on LHS (e.g. `*it = x;`)
    - Some can be decremented `--`
    - Some support random access (`[[]], +, -, +=, -=, <, >` operators)
#include <vector>

#include "Tracer.h"

using namespace std;

int main(int argc, char** argv) {
    Tracer a, b, c;
    vector<Tracer> vec;

    vec.push_back(a);
    vec.push_back(b);
    vec.push_back(c);

    cout << "Iterating:" << endl;
    vector<Tracer>::iterator it;
    for (it = vec.begin(); it < vec.end(); it++) {
        cout << *it << endl;
    }
    cout << "Done iterating!" << endl;
    return EXIT_SUCCESS;
}
Type Inference (C++11)

- The **auto** keyword can be used to infer types
  - Simplifies your life if, for example, functions return complicated types
  - The expression using **auto** must contain explicit initialization for it to work

```cpp
// Calculate and return a vector containing all factors of n
std::vector<int> Factors(int n);

void foo(void) {
    // Manually identified type
    std::vector<int> facts1 = Factors(324234);

    // Inferred type
    auto facts2 = Factors(12321);

    // Compiler error here
    auto facts3;
}
```
auto and Iterators

- Life becomes much simpler!

```cpp
for (vector<Tracer>::iterator it = vec.begin(); it < vec.end(); it++) {
    cout << *it << endl;
}
```

```cpp
for (auto it = vec.begin(); it < vec.end(); it++) {
    cout << *it << endl;
}
```
Range for Statement (C++11)

- Syntactic sugar similar to Java’s foreach

```cpp
for ( declaration : expression ) {
    statements
}
```

- `declaration` defines loop variable
- `expression` is an object representing a sequence
  - Strings, initializer lists, arrays with an explicit length defined, STL containers that support iterators

```cpp
// Prints out a string, one character per line
std::string str("hello");
for ( auto c : str ) {
    std::cout << c << std::endl;
}
```
Updated iterator Example

```cpp
#include <vector>
#include "Tracer.h"

using namespace std;

int main(int argc, char** argv) {
    Tracer a, b, c;
    vector<Tracer> vec;

    vec.push_back(a);
    vec.push_back(b);
    vec.push_back(c);

    cout << "Iterating:" << endl;
    // "auto" is a C++11 feature not available on older compilers
    for (auto& p : vec) {
        cout << p << endl;
    }
    cout << "Done iterating!" << endl;
    return EXIT_SUCCESS;
}
```
STL Algorithms

- A set of functions to be used on ranges of elements
  - **Range**: any sequence that can be accessed through *iterators* or *pointers*, like arrays or some of the containers
  - General form: `algorithm(begin, end, ...);

- Algorithms operate directly on range *elements* rather than the containers they live in
  - Make use of elements’ copy ctor, =, ==, !=, <
  - Some do not modify elements
    - e.g. `find`, `count`, `for_each`, `min_element`, `binary_search`
  - Some do modify elements
    - e.g. `sort`, `transform`, `copy`, `swap`
#include <vector>
#include <algorithm>
#include "Tracer.h"
using namespace std;

void PrintOut(const Tracer& p) {
    cout << " printout: " << p << endl;
}

int main(int argc, char** argv) {
    Tracer a, b, c;
    vector<Tracer> vec;

    vec.push_back(c);
    vec.push_back(a);
    vec.push_back(b);
    cout << "sort:" << endl;
    sort(vec.begin(), vec.end());
    cout << "done sort!" << endl;
    for_each(vec.begin(), vec.end(), &PrintOut);
    return 0;
}
LLVM

Eigen template matrix

boost template library

Bonus: `unique_ptr`
std::unique_ptr

- A `unique_ptr` takes ownership of a pointer
  - Part of C++’s standard library (C++11)
  - A template: template parameter is the type that the “owned” pointer references (i.e. the `T` in pointer type `T*`)
  - Its destructor invokes `delete` on the owned pointer
    - Invoked when `unique_ptr` object is `delete`’d or falls out of scope
#include <iostream> // for std::cout, std::endl
#include <memory> // for std::unique_ptr
#include <cstdlib> // for EXIT_SUCCESS

void Leaky() {
    int *x = new int(5); // heap-allocated
    (*x)++;
    std::cout << *x << std::endl;
} // never used delete, therefore leak

void NotLeaky() {
    std::unique_ptr<int> x(new int(5)); // wrapped, heap-allocated
    (*x)++;
    std::cout << *x << std::endl;
} // never used delete, but no leak

int main(int argc, char **argv) {
    Leaky();
    NotLeaky();
    return EXIT_SUCCESS;
}
Why are `unique_ptr` useful?

- If you have many potential exits out of a function, it’s easy to forget to call `delete` on all of them
  - `unique_ptr` will `delete` its pointer when it falls out of scope
  - Thus, a `unique_ptr` also helps with exception safety

```cpp
void NotLeaky() {
    std::unique_ptr<int> x(new int(5));
    ...
    // lots of code, including several returns
    // lots of code, including potential exception throws
    ...
}
```
#include <memory>  // for std::unique_ptr
#include <cstdlib>  // for EXIT_SUCCESS

using namespace std;
typedef struct { int a, b; } IntPair;

int main(int argc, char **argv) {
    unique_ptr<int> x(new int(5));

    int *ptr = x.get();  // Return a pointer to pointed-to object
    int val = *x;        // Return the value of pointed-to object

    // Access a field or function of a pointed-to object
    unique_ptr<IntPair> ip(new IntPair);
    ip->a = 100;

    // Deallocate current pointed-to object and store new pointer
    x.reset(new int(1));

    ptr = x.release();  // Release responsibility for freeing
    delete ptr;
    return EXIT_SUCCESS;
}
**unique_ptr**s Cannot Be Copied

- **std::unique_ptr** has disabled its copy constructor and assignment operator
  - You cannot copy a **unique_ptr**, helping maintain “uniqueness” or “ownership”

```cpp
#include <memory>  // for std::unique_ptr
#include <cstdlib> // for EXIT_SUCCESS

int main(int argc, char **argv) {
  std::unique_ptr<int> x(new int(5));  //

  std::unique_ptr<int> y(x);     //
  std::unique_ptr<int> z;        //
  z = x;                        //

  return EXIT_SUCCESS;
}
```
Transferring Ownership

- Use \texttt{reset()} and \texttt{release()} to transfer ownership
  - \texttt{release} returns the pointer, sets wrapped pointer to \texttt{nullptr}
  - \texttt{reset} deletes the current pointer and stores a new one.

```cpp
int main(int argc, char **argv) {
    unique_ptr<int> x(new int(5));
    cout << "x: " << x.get() << endl;

    unique_ptr<int> y(x.release()); // x abdicates ownership to y
    cout << "x: " << x.get() << endl;
    cout << "y: " << y.get() << endl;

    unique_ptr<int> z(new int(10));

    // y transfers ownership of its pointer to z.
    // z's old pointer was delete'd in the process.
    z.reset(y.release());

    return EXIT_SUCCESS;
}
```
unique_ptr and Arrays

- **unique_ptr** can store arrays as well
  - Will call `delete[]` on destruction

```cpp
#include <memory>  // for std::unique_ptr
#include <cstdlib>  // for EXIT_SUCCESS

using namespace std;

int main(int argc, char **argv) {
  unique_ptr<int[]> x(new int[5]);

  x[0] = 1;
  x[2] = 2;

  return EXIT_SUCCESS;
}
```