Lecture 24: C++

Inheritance
Administrivia

- HW 3 posted Friday -> Extra credit due date Wednesday Nov 25th @ 9pm
- HW 4 posted Tuesday -> Extra credit due date Wednesday
- End of quarter due date Wednesday December 16th @ 9pm
```cpp
#ifndef BANKACCOUNT_H
#define BANKACCOUNT_H

#include <iostream>

namespace bank {

class BankAccount {
public:
  explicit BankAccount(const std::string& accountHolder);
  BankAccount(const BankAccount& other) = delete;

  // Accessors
  int getBalance() const;
  int getAccountId() const;
  const std::string& getAccountHolder() const;

  // Modifier - add money.
  void deposit(int amount);

  // different for every type of account,
  // require derived classes to implement
  virtual void withdraw(int amount) = 0;

protected:
  // derived classes can modify the balance.
  void setBalance(int balance);

private:
  const std::string accountHolder_;  
  const int accountId_; 
  int balance_; 

  static int accountCount_; 
};
}
#endif

#ifndef SAVINGSACCOUNT_H
#define SAVINGSACCOUNT_H

#include "BankAccount.h"

namespace bank {

class SavingsAccount : public BankAccount {
public:
  SavingsAccount(double interestRate, std::string name);

  double getInterestRate() const;
  virtual void withdraw(int amount) override;

private:
  bool isNewMonth(time_t* curTime);

  double interestRate_; 
  time_t lastMonth_; 
  int numTransactionsInMonth_; 
};
}
#endif

BankAccount.cc

SavingsAccount.cc
```
#include <iostream>

using namespace std;

class A {
public:
    A() { cout << "a()" << endl; }
    ~A() { cout << "~a" << endl; }
    void m1() { cout << "a1" << endl; }
    void m2() { cout << "a2" << endl; }
};

// class B inherits from class A
class B : public A {
public:
    B() { cout << "b()" << endl; }
    ~B() { cout << "~b" << endl; }
    void m2() { cout << A::m2();
                << "b2" << endl; }
    void m3() { cout << "b3" << endl; }
};

int main() {
    //B* x = new B();
    A* x = new B();
    x->m1();
    x->m2();
    x->m3();
    delete x;
}
Suppose that...

- You want to write a function to compare two ints
- You want to write a function to compare two strings
  - Function overloading!
- 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

```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;
}
```
Polymorphism in C++

- **In Java:** `PromisedType var = new ActualType()``
  - `var` is a reference (different term than C++ reference) to an object of `ActualType` on the Heap
  - `ActualType` must be the same class or a subclass of `PromisedType`

- **In C++:** `PromisedType* var_p = new ActualType();``
  - `var_p` is a `pointer` to an object of `ActualType` on the Heap
  - `ActualType` must be the same or a derived class of `PromisedType`
  - (also works with references)
  - `PromisedType` defines the *interface* (i.e. what can be called on `var_p`), but `ActualType` may determine which *version* gets invoked

- Polymorphism is the ability to access different objects through the same interface
Templates in C++

- 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 Template

// 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;
}

#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;
}
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

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

// returns 0 if equal, 1 if value1 is bigger, // -1 otherwise
template <typename 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
    return EXIT_SUCCESS;
}
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 and get the value of, but we don’t know what data type the things will be

- 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)

```cpp
#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 Definition

```cpp
#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;
}
```

```cpp
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() << \
    ");
}
```
Abstract Methods & Classes

- Sometimes we want to include a function in a class but only implement it in derived classes
  - In Java, we would use an abstract method
  - In C++, we use a “pure virtual” function
    - Example: virtual string noise() = 0;

- virtual string noise() = 0;

- A class containing any pure virtual methods is abstract
  - You can’t create instances of an abstract class
  - Extend abstract classes and override methods to use them

- A class containing only pure virtual methods is the same as a Java interface
  - Pure type specification without implementations
Virtual Functions

- A **virtual function** is a member function that is declared within a base class and is overridden by a derived class,
  - Ensures correct function is called for object regardless of reference type (facilitate polymorphism)
  - A method-call is virtual if the method called is marked virtual or overrides a virtual method
  - a non-virtual method call is resolved using the compile-time type of the receiver expression
  - a virtual method call is resolved using the run-time class of the receiver object (what the expression evaluates to) AKA: dynamic dispatch

- **pure virtual functions**
  - to maximize code sharing sometimes you will need “theoretical” objects or functions that will be shared across more specific implementations. (EX: “bank account” is too general to exist, instead you use it to share code across “checking account” and “business account”)
  - When defining abstract classes sometimes you want to declare a function that must be implemented by all derived classes, you can create a virtual function:
    - virtual void withdraw(int amount) = 0;
    ```cpp
class C {
  virtual void withdraw(int amount) = 0;
  virtual t0 m(t1, t2,...,tn) = 0;
  ...
};
```
Dynamic Dispatch

- **Dynamic dispatch** is the process of selecting which implementation of a polymorphic operation to call at runtime.

- Usually, when a derived function is available for an object, we want the derived function to be invoked.
  - This requires a *run time* decision of what code to invoke.

- A member function invoked on an object should be the *most-derived function* accessible to the object’s visible type.
  - Can determine what to invoke from the *object* itself.

- **Example:**
  - `void PrintStock(Stock* s) { s->Print(); }`

- Calls the appropriate `Print()` without knowing the actual type of `*s`, other than it is some sort of `Stock`.

- Functions just like Java.

- Unlike Java: Prefix the member function declaration with the virtual keyword.
  - Derived/child functions don’t need to repeat virtual, but was traditionally good style to do so.
  - This is how method calls work in Java (no virtual keyword needed).
  - You almost always want functions to be virtual.
#include "Stock.h"
#include "DividendStock.h"

DividendStock dividend();
DividendStock* ds = &dividend;
Stock* s = &dividend;  // why is this allowed?

// Invokes DividendStock::GetMarketValue()
//   since that method is inherited.
// Stock::GetProfit() invokes
//   since that is the most-derived accessible function.

s->GetProfit();
Most-Derived Self-Check

class A {
  public:
    virtual void Foo();
};

class B : public A {
  public:
    virtual void Foo();
};

class C : public B {
};

class D : public C {
  public:
    virtual void Foo();
};

class E : public C {
};

void Bar() {
  A* a_ptr;
  C c;
  E e;

  // Q1:
  a_ptr = &c;
  a_ptr->Foo();

  // Q2:
  a_ptr = &e;
  a_ptr->Foo();
}

Q1  Q2
A.  A  B
B.  A  D
C.  B  B
D.  B  D
How does dynamic dispatch work?

- The compiler produces Stock.o from just Stock.cc
  - It doesn’t know that DividendStock exists during this process
  - So then how does the emitted code know to call Stock::\texttt{GetMarketValue()} or DividendStock::\texttt{GetMarketValue()} or something else that might not exist yet?
    - \textit{Function pointers}!!!

\begin{verbatim}
Stock.h

virtual double Stock::\texttt{GetMarketValue()} const;
virtual double Stock::\texttt{GetProfit()} const;

\end{verbatim}

\begin{verbatim}
Stock.cpp

double Stock::\texttt{GetMarketValue()} const {
    return get_shares() * get_share_price();
}

double Stock::\texttt{GetProfit()} const {
    return GetMarketValue() - GetCost();
}
\end{verbatim}
vtables and vptrs

- If a class contains *any* virtual methods, the compiler emits:
  - A (single) virtual function table (vtable) for the class
    - Contains a function pointer for each virtual method in the class
    - The pointers in the vtable point to the most-derived function for that class
  - A virtual table pointer (vptr) for each object instance
    - A pointer to a virtual table as a “hidden” member variable
    - When the object’s constructor is invoked, the vptr is initialized to point to the vtable for the object’s class
    - Thus, the vptr “remembers” what class the object is
Dynamic Dispatch Visual

Dynamic Dispatch

Point object

Point vtable:

3DPoint object

3DPoint vtable:

Java:
```
Point p = ???;
return p.samePlace(q);
```

C pseudo-translation:
```
// works regardless of what p is
return p->vtable[1](p, q);
```
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!
**C++ Standard Libraries**

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

```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;
}
```
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)

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

```cpp
#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;
}
```
Questions
"Resource Acquisition is Initialization"

Design pattern at the core of C++

When you create an object, acquire resources
- Create = constructor
- Acquire = allocate (e.g. memory, files)

When the object is destroyed, release resources
- Destroy = destructor
- Release = deallocate

When used correctly, makes code safer and easier to read

```c
char* return_msg_c() {
    int size = strlen("hello") + 1;
    char* str = malloc(size);
    strncpy(str, "hello", size);
    return str;
}
```

```cpp
std::string return_msg_cpp() {
    std::string str("hello");
    return str;
}
```

```c
using namespace std;
char* s1 = return_msg_c();
cout << s1 << endl;
string s2 = return_msg_cpp();
cout << s2 << endl;
```
RAII Example

- Which do you prefer?
- Where is the bug?

```c
char* return_msg_c() {
    int size = strlen("hello") + 1;
    char* str = malloc(size);
    strncpy(str, "hello", size);
    return str;
}
```

```cpp
std::string return_msg_cpp() {
    std::string str("hello");
    return str;
}
```

```cpp
using namespace std;
char* s1 = return_msg_c();
cout << s1 << endl;
string s2 = return_msg_cpp();
cout << s2 << endl;
```
The compiler sometimes uses a “return by value optimization” or “move semantics” to eliminate unnecessary copies.
- Sometimes you might not see a constructor get invoked when you might expect it.

```cpp
Point foo() {
    Point y;  // default ctor
    return y;  // copy ctor? optimized?
}
Point x(1, 2);  // two-ints-argument ctor
Point y = x;  // copy ctor
Point z = foo();  // copy ctor? optimized?
```
Namespaces

- Each namespace is a separate scope
  - Useful for avoiding symbol collisions!

- Namespace definition:
  - `namespace name {
    // declarations go here
  }
  - Doesn’t end with a semi-colon and doesn’t add to the indentation of its contents
  - Creates a new namespace name if it did not exist, otherwise *adds to the existing namespace* (!)
    - This means that components (e.g. classes, functions) of a namespace can be defined in multiple source files

- Namespaces vs classes
  - They seem somewhat similar, but classes are *not* namespaces:
  - There are no instances/objects of a namespace; a namespace is just a group of logically-related things (classes, functions, etc.)
  - To access a member of a namespace, you must use the fully qualified name (i.e. `nsp_name::member`)
    - Unless you are using that namespace
    - You only used the fully qualified name of a class member when you are defining it outside of the scope of the class definition
Const

- C++ introduces the “const” keyword which declares a value that cannot change
- const int CURRENT_YEAR = 2020;