C++ Overview (1)

COS320
Heejin Ahn
(heejin@cs.princeton.edu)
Introduction

• Created by Bjarne Stroustrup

• Standards
  – C++98, C++03, C++07, C++11, and C++14

• Features
  – Classes and objects
  – Operator overloading
  – Templates
  – STL (Standard Template Library)
  – ...

• Still widely used in performance-critical programs

• This overview assumes that you know C and Java
C++ is a Federation of Languages

- C
  - Mostly backward compatible with C
  - Blocks, statements, preprocessor, built-in data types, arrays, pointers, ...

- Object-Oriented C++
  - Classes, encapsulation, inheritance, polymorphism, virtual functions, ...

- Template C++
  - Paradigm for generic programming

- STL (Standard Template Library)
  - Generic library using templates
  - Containers, iterators, algorithms, function objects ...
Topics

• Today
  – Heap memory allocation
  – References
  – Classes
  – Inheritance

• Next time
  – Operator overloading
  – I/O streams
  – Templates
  – STL
  – C++11
Heap allocation: new and delete

- new/delete is a type-safe alternative to malloc/free
- new T allocates an object of type T on heap, returns pointer to it
  - Stack *sp = new Stack();
- new T[n] allocates array of T's on heap, returns pointer to first
  - int *stk = new int[100];
  - By default, throws exception if no memory
- delete p frees the single item pointed to by p
  - delete sp;
- delete[] p frees the array beginning at p
  - delete[] stk;
- new uses T's constructor for objects of type T
  - need a default constructor for array allocation
- delete uses T's destructor ~T()
- use new/delete instead of malloc/free and never mix new/delete and malloc/free
References

• Controlled pointers

• When you need a way to access an object, not a copy of it
• In C, we used pointers
  – `int var = 3;`
  – `int *pvar = &var;`
  – `*pvar = 5; // now var == 5`
• In C++, references attach a name to an object
  – `int var = 3;`
  – `int &rvar = var;`
  – `rvar = 5; // now var == 5`

• Unlike pointers, you can’t define a reference without an object it refers to
  – `int &x; (X)`
Call-by-Reference

• Call-by-reference allows you to modify arguments
  – Now you can implement swap() function using call-by-reference

```c
void swap(int *x, int *y) {
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}

swap (&a, &b);
```

```c
void swap(int &x, int &y) {
    int temp;
    temp = x;
    x = y;
    y = temp;
}

// pointers are implicit
swap(a, b);
```
Call-by-Value / Call-by-Reference

• Call-by-value
  – By default, C/C++’s uses call-by-value
  – If you pass an object using call-by-value, it causes the object to copy, which is inefficient if it is large

• Call-by-reference
  – In effect, you just pass the address of an object
  – Call-by-const-reference additionally guarantees that the object will not be modified during the call
  – Java function call is similar to call-by-reference
    • Java actually passes pointers internally

```c++
// call-by-reference
void foo(Stack &s) {
    ...
}
Stack s;
foo(s); // s can be modified in foo

// call-by-const-reference
void foo(const Stack &s) {
    ...
}
Stack s;
foo(s); // s is guaranteed to stay the same
```
Constness

• Way to say something is not modifiable

char greeting[] = "Hello";
char *p = greeting; // non-const pointer, non-const data
const char *p = greeting; // non-const pointer, const data
char * const p = greeting; // const pointer, non-const data
const char * const p = greeting; // const pointer, const data

// Objects ‘a’ references or ‘b’ points to cannot be modified in this function
void foo(const Stack &a, const Stack *b);

// For class member member method
// This does not modify any status of ‘this’ object
void Stack::size() const;
C++ Classes

- defines a data type 'Thing'
  - can declare variables and arrays of this type, create pointers to them, pass them to functions, return them, etc.
- Object: an instance of a class variable
- Method: a function defined within the class

```cpp
class Thing {
public:
    methods and variables accessible from all classes
protected:
    methods and variables accessible from this class and child classes
private:
    methods and variables only visible to this class
};
```

Don’t miss ‘;’ from [1]
// simple-minded stack class
class Stack {
public:
  Stack(); // constructor decl
  int push(int);
  int pop();
private: // default visibility
  int stk[100];
  int *sp;
};

Stack::Stack() { // constructor implementation
  sp = stk;
}
int stack::push(int n) {
  return *sp++ = n;
}
int stack::pop() {
  return *--sp;
}

Stack s1, s2; // calls constructors
s1.push(1); // method calls
s2.push(s1.pop());
Constructors

• Creates a new object
• Construction includes initialization, so in may be parameterized like other methods
  – You can have multiple constructors with different parameters
• If you don’t define any constructors, a default constructor will be generated
  – Student() {}
• ‘this’ is a pointer – so you need ‘->’ to refer to it

class Student {
public:
    Student(const string &n, double gpa) {
        name = n;
        this->gpa = gpa; // when a member variable name and a parameter name are the same
    }
private:
    string name;
    double gpa;
};
Constructors

class MyClass {
public:
    MyClass(int arg) { ... }
    MyClass(int arg1, arg2) { ... }
    ... 
};

// These call constructor and create objects on the stack
MyClass m1(100);
MyClass m2 = 100;
MyClass m3(100, 300);

// These call constructor and create objects on the heap
MyClass *m4 = new MyClass(100);
MyClass *m5 = new MyClass(100, 300);

// You can omit () when you call the default constructor (if there is one)
MyClass m6; // equivalent to ‘MyClass m6();’
MyClass *m7 = new MyClass; // equivalent to ‘MyClass *m7 = new MyClass();’
‘explicit’ Keyword

• Compiler does automatic type-conversion for one-argument constructors
  – When this is a constructor:
    • MyClass(int arg);
  – This creates a temporary object and assigns it to m
    • MyClass m = 5;
• It is sometimes not what we want; to prevent this, use ‘explicit’ keyword
  – explicit MyClass(int arg);

```cpp
// When this can be desirable
class string {
public:
    string(const char *s);
};

string str = “Hello world”; // Good!
```

```cpp
// When this is not desirable
class vector {
public:
    vector(int size);
};

vector v = 5; // Oh no! Use ‘explicit’
```
Inline Functions

• Inlined functions are copied into callers’ body when compiled
  – Can prevent function call overhead
  – Can increase code size
• ‘inline’ directive suggests the function is to be inlined
  – So inline functions should be available at compile time (not link time) –
    they should exist in the same source file or any other header files included
  – But the final inlining decision is on compiler
• Member functions defined within a class have the same effect

```cpp
class Student {
public:
    void setGPA(double gpa) {
        this->gpa = gpa;
    }
    ...
};
```

```cpp
class Student {
public:
    void setGPA(double gpa);
    ...
};
inline void setGPA(double gpa) {
    this->gpa = gpa;
}
```
Accessors vs. Mutators

• Mutators can alter the state (= non-static member variables) of an object, while accessors cannot

• Accessors have ‘const’ at the end of the signature

```cpp
class Student {
public:
    ...
    void setGPA(double gpa) { this->gpa = gpa; } // mutator
    double getGPA() const { return gpa; } // accessor
    ...
};
```
Initializer Lists

- Initialization lists are more efficient
  - Only call constructors once
  - Difference is small for primitive data types
- You have to use initializer lists for some variables
  - Class objects that do not have default (= no argument) constructors
  - Reference variables
    - They cannot be created without the target they refer to
Default Parameters

• Specifies default values for function parameters
• Included in the function declaration

```c
void printInt(int n, int base=10);

printInt(50); // equivalent to ‘printInt(5, 10);’, outputs 50
printInt(50, 8); // outputs 62 (50 in octal)
```
The Big Three

• Copy Constructor
  – MyClass(const MyClass &rhs)
  – Special constructor to construct a new object as a copy of the same type of object

• operator=
  – operator=(const MyClass &rhs)
  – Copy assignment operator
  – Applied to two already constructed objects

• Destructor
  – ~MyClass()
  – Destroys an existing object
  – If you have some member variables that are ‘new’ed, you should delete them here
  – Called when
    • An object on the stack goes out of scope ()
    • An object on the heap is deleted using ‘delete’

• If you don’t write these by yourself, default ones will be generated by compiler
  – Write these only if you want to do some additional tasks

from [2]
class Student {
public:
  // Normal constructor
  Student(const string &name, double gpa) : name(name), gpa(gpa) {
    someObj = new MyClass();
  }

  // Copy constructor
  Student(const Student &rhs) : name(rhs.name), gpa(rhs.gpa) {
    someObj = new MyClass();
  }

  // Destructor
  ~Student() { delete someObj; }

  // operator=
  operator=(const Student &rhs) {
    name = rhs.name;
    gpa = rhs.gpa;
    someObj = MyClass(rhs.someObj); // Calls copy constructor of MyClass
  }

private:
  string name;
  double gpa;
  MyClass *someObj;
};
The Big Three

```cpp
void someFunction() {

    // Calls the normal constructor
    Student jane("Jane", 3.0); // stack
    Student *pJane = new Student("Jane", 3.0); // heap

    // Calls the copy constructor
    Student tom(jane); // stack
    student *pTom = new Student(jane); // heap

    // Calls the operator=()
    tom = jane;
    *pTom = *pJane;

    // Calls the destructor for heap objects
    delete pJane;
    delete pTom;

} // At this point the destructor for stack objects (jane and tom) are called
```
**Friends**

- Way to grant private member access to specific classes/functions

```cpp
class ListNode {
private:
    int element;
    ListNode *next;
    ListNode(int element, ListNode *next=NULL) : element(element), next(next) {}

    friend class List; // friend class
    friend int someFunction(); // friend methods
};

class List {
public:
    List() {
        head = new ListNode(); // can call ListNode’s private constructor
    }
private:
    ListNode *head;
};

int someFunction() { ... }
```
The struct Type

• A class in which the members default to public
  – In a class, the members default to private
• Unlike C, you don’t need a ‘struct’ keyword to refer to a struct type, because it is now a ‘class’

```c
struct MyClass {
    MyClass(int arg) { ... }
    ... 
};

MyClass m(100);
```
Namespaces

• C++ equivalent of Java packages

```cpp
namespace myspace {
    class Student { ... };
    class Professor { ... };
}

// Refers to student class with the namespace name
myspace::Student s;
myspace::Professor p;

// This allows you to use ‘Student’ without the namespace name
using myspace::Student;
Student s;
myspace::Professor p;

// This allows you to use everything in myspace without the namespace name
using myspace;
Student s;
Professor p;
```
Incomplete Class Declaration

• Unlike Java, the order of class declaration matters
  – If class B is declared after class A, class A does not know about class B
  – But sometimes class A needs to know (at least) the existence of class B
    • ex) A has B’s pointer as a member and B has A’s pointer too

• Solution: incomplete class declaration
  – We can incompletely declare class B before class A
  – The only thing class A knows about B is its existence; A does not know about B’s members or B’s object size because B’s full declaration is below A
### Incomplete Class Declaration

```cpp
class B; // incomplete class declaration

class A { // Now A knows B’s existence
public:
    void foo1(B *b); // OK
    void foo2(B &b); // OK
    void foo3(B b); // (X) Not OK! A does not know B’s object size

    void bar(B *b) {
        b->baz(); // (X) Not OK! A does not know about B’s members
    }

    B *data1; // OK
    B &data2; // OK
    B data3; // (X) Not OK! A does not know B’s object size
};

class B {
public:
    void baz() { ... }
    A *data;
};
```
Inheritance

• Basic syntax
  – class Child : public Parent { ... };
  – Actually there are also private and protected inheritance – nevermind, you are not going to use them. Just don’t forget to use ‘public’ keyword

• Calling base class constructor in initializer lists
  – Child(int arg1, int arg2)
    : Parent(arg1), ... { ... }

• Calling base class functions
  – Parent::foo(...);
Inheritance

class Person {
public:
    Person(int ssn, const string &name) : ssn(ssn), name(name) {}
    const string &getName() const { return name; }
    int getSSN() const { return ssn; }
    void print() const { cout << ssn << "", " " << name; }

private:
    int ssn;
    string name;
};

class Student : public Person {
public:
    Student(int ssn, const string &name, double gpa) : Person(ssn, name), gpa(gpa) {} 
    double getGPA() const { return gpa; }
    void print() const { Person::print(); cout << "", " " << gpa; } // override

private:
    double gpa;
};
Dynamic Dispatch

• Java always uses the runtime type to decide which method to use
• But C++ uses the static type by default

Student s(123456789, "Jane", 4.0);
s.print(); // calls Student::print

Person &p1 = s; // p1 and s are same object
p1.print(); // calls Person::print!

Person *p2 = &s; // p2 points to s
p2->print(); // calls Person::print!
Virtual Functions

• You need a ‘virtual’ keyword to tell the compiler this function should use dynamic dispatch
• In Java, all functions are virtual functions

```cpp
class Person {
public:
    ...  
    virtual void print() const { cout << ssn << "\", " << name }
    ...  
};

class Student : public Person {
public:
    ...  
    virtual void print() const { Person::print(); cout << "\", " << gpa; }
    ...  
};
```
This can be omitted in child class
The Big Three Revisited

• In a subclass, the default Big Three are constructed if you don’t explicitly define them

• Copy constructor
  – Invokes copy constructor on the base class(es)
  – And then invokes copy constructors on each of newly added members

• operator=
  – Invokes operator= on the base class(es)
  – And then invokes operator= on each of newly added members

• Destructor
  – Invokes destructors on each of newly added members
  – And then invokes destructor on the base class(es)
Virtual Destructor

- In a base class, the destructor should be declared virtual
  - Otherwise the base class portion of the object will not be deleted if it is deleted through base class pointer/reference

```cpp
class Person {
public:
    ...
    virtual ~Person() { ... }
    ...
};

Student *tom = new Student("123456", "Tom", 3.0);
Person *p = tom;

// If ~Person() is declared non-virtual, this will not delete the base class portion of the object
delete p;
```
Abstract Methods and Classes

• C++ equivalent of Java abstract methods and classes
  – Abstract classes cannot be instantiated

• In C++, a method is abstract if:
  – It is declared virtual
  – The declaration is followed by =0
  – ex) virtual double area() const = 0;

• In C++, a class is abstract if:
  – It has at least one abstract method
Slicing

• You can access child classes by pointers and references of base classes
• But you can’t use base classes objects to access child classes

Student jane(12345, “Jane”, 4.0);
Person p(54321, “Bob”);
p = jane; // object is sliced!
p.print();

void print(Person p) { p.print(); }
print(jane); // call-by-value, so object is sliced!
Type Conversions

• C++-style casts
  – static_cast
    • Similar to C-style cast
  – dynamic_cast
    • Can only be used with pointers and references to classes (or with void*)
    • Performs a runtime check if the cast is correct; returns NULL if incorrect
  – const_cast
    • Manipulates the constness of the object pointed by a pointer, either to be set or to be removed
  – reinterpret_cast
    • Converts any pointer type to any other pointer type
    • Simple binary copy of the value from one pointer to the other

• LLVM-style casts
  – cast, dyn_cast, cast_or_null, dyn_cast_or_null
class Base {};
class Derived: public Base {};

Base *pb = new Base;

// C-style cast
Derived *pd = (Derived*) pb;

// static_cast
Derived *pd = static_cast<Derived*>(pb);

// dynamic_cast
Derived *pd = dynamic_cast<Derived*>(pb); // returns NULL if failed
if (!pd) cout << "Type casting failed!";

// const_cast
void printStr (char *str) {
    cout << str << '\n';
}
const char *c = "Hello world";
printStr(const_cast<char*>(c)); // removes constness to pass it to printStr
Multiple Inheritance

• Don’t use this unless you really need to...
  – Has many tricky aspects
  – You don’t need to use this in your code in COS320

• We are not going to cover the details here

```cpp
class Person : public Printable, public Serializable { ... };
class Student : virtual public Person { ... };
class Employee : virtual public Person { ... };
class StudentEmployee : public Student, public Employee { ... };
```

from [2]
References