

CSC 533: Organization of Programming Languages

Spring 2007

Java vs. C++

- C++ design goals
- C++ reliability features
 - by-reference, const, new & delete, bool, string
- C++ OOP features
 - classes, bindings, templates, inheritance
- C++ examples
- Java design goals

1

C++ design

C was developed by Dennis Ritchie at Bell Labs in 1972

- designed as an in-house language for implementing UNIX
- provided many high-level features (control structures, functions, arrays, structures)
- also some low-level features (address-of operator, memory allocation/deallocation)
- somewhat kludgy: weak type checking, reliance on preprocessor, ...

C++ was developed by Bjarne Stroustrup at Bell Labs in 1984

- C++ is a superset of C, with language features added to support OOP

design goals:

1. support object-oriented programming (i.e., classes & inheritance)
2. retain the high performance of C
3. provide a smooth transition into OOP for procedural programmers

2

C++ design

backward compatibility with C was key to the initial success of C++

- programmers could continue to use existing C code
- could learn and add new features incrementally

however, backward compatibility had far-reaching ramifications

- C++ did add many features to improve reliability & support OOP
- but, couldn't remove undesirable features
 - it is a large, complex, and sometimes redundant language

features that improved reliability:

- by-reference parameter passing
- constants
- new & delete
- bool & string

3

Added reliability features: pass by-reference

in C, all parameter passing was by-value

```
void reset(int num) {           |           int x = 9;
    num = 0;                   |           reset(x);
}                               |           printf("x = %d", x);
```

- but, could get the effect of by-reference via pointers

```
void reset(int* num) {         |           int x = 9;
    *num = 0;                  |           reset(&x);
}                               |           printf("x = %d", x);
```

C++ introduced cleaner by-reference passing (in addition to default by-value)

```
void reset(int & num) {        |           int x = 9;
    num = 0;                   |           reset(x);
}                               |           cout << "x = " << x;
```

4

Added reliability features: constants

in C, constants had to be defined as preprocessor directives

- weakened type checking, made debugging more difficult

```
#define MAX_SIZE 100
```

C++ introduced the `const` keyword

- can be applied to constant variables (similar to `final` in Java)
the compiler will catch any attempt to reassign

```
const int MAX_SIZE = 100;
```

- can also be applied to by-reference parameters to ensure no changes
safe since `const`; efficient since by-reference

```
void process(const ReallyBigObject & obj) {  
    . . .  
}
```

5

Other reliability features

in C, memory was allocated & deallocated using low-level system calls

- C++ introduced typesafe operators for allocating & deallocating memory

```
int* a = (int*)malloc(20*sizeof(int));    int* a = new int[20];  
...                                       ...  
free(a);                                  delete[] a;
```

in C, there was no boolean type – had to rely on user-defined constants

- C++ `bool` type still implemented as an int, but provided some level of abstraction

```
#define FALSE 0                                bool flag = true;  
#define TRUE 1  
  
int flag = TRUE;
```

in C, there was no string type – had to use char arrays & library functions

- C++ `string` type encapsulated basic operations inside a class

```
char* word = "foo";                            string word = "foo";  
printf("%d", strlen(word));                    cout << word.length();
```

6

C++ classes

C++ classes were inspired by Simula67, the first OOP language

- but followed the structure of C structs (records)

```
struct Point {
    int x;
    int y;
};

struct Point pt;
pt.x = 3;
pt.y = 4;
```

```
class Point {
public:
    Point(int xCoord, int yCoord) {
        x = xCoord; y = yCoord;
    }

    int getX() const { return x; }

    int getY() const { return y; }

private:
    int x;
    int y;
};

Point pt(3, 4);
```

again, for backward compatibility, structs remained in C++

only difference: in a struct, fields/functions are `public` by default
in a class, fields/functions are `private` by default

7

Procedural vs. OOP

C++ is considered a hybrid language since

- can write a program procedurally: a collection of stand-alone functions
- can write a program object-oriented: a collection of interacting classes
- can mix the two approaches: some classes & some stand-alone functions

to maintain the efficiency of C function calls, function/member function calls are bound statically

- this has impact on inheritance & polymorphism
- must explicitly declare member functions to be virtual to get full effect

```
class Person {
    ...
};

class Student : public Person {
    ...
};
```

```
void stuff(Person p) {
    p.Display();
}

Student s1(...); // by default, will call
stuff(s1);       // Person::Display
                // must declare Person::Display
                // to be virtual to force
                // Student::Display
```

8

Memory management

by default, variables in C++ are bound to memory stack-dynamically

- allocated when declaration is reached, stored on the stack
- this includes instances of classes as well as primitives

can use `new` & `delete` to create heap-dynamic memory

- requires diligence on the part of the programmer
- must explicitly delete any heap-dynamic memory, or else garbage references persist (there is no automatic garbage collection)
- in order to copy a class instance with heap-dynamic fields, must define a special copy constructor
- in order to reclaim heap-dynamic fields, must define a special destructor

9

Templated classes & functions

in C++ can parameterize classes/functions using templates

- recall, this feature was added to Java 5.0

```
template <class Type>
class MyList {
public:
    . . .
private:
    <Type> items[];
};
```

must specify `Type` when declare an object

```
MyList<int> nums(20);
```

```
template <class Item>
void Display(Item x)
{
    cout << x << endl;
}
```

when called, `Item` is automatically instantiated (must support `<<`)

```
Date day(9, 27, 2000);
Display(day);
```

C++ provides the Standard Template Library (STL), a collection of templated classes similar to Java's libraries

10

Inheritance

inheritance in C++ looks similar to Java

```
public class Student extends Person {
    public Student(String nm, String id,
                  char sex, int yrs,
                  String sch, int lvl) {
        super(nm, id, sex, yrs);
        school = sch; grade = lvl;
    }

    public void Advance() {
        grade++;
    }

    public void Display() {
        super.Display();
        System.out.println("School: " + school +
                          "\nGrade: " + grade);
    }

    private String school;
    private int grade;
}
```

```
class Student : public Person {
public:
    Student(string nm, string id, char sex,
           int yrs, string sch, int lvl) :
        Person(nm, id, sex, yrs) {
        school = sch; grade = lvl;
    }

    void Advance() {
        grade++;
    }

    void Display() {
        Person::Display();
        cout << "School: " << school
             << endl << "Grade: "
             << grade << endl;
    }

private:
    string school;
    int grade;
};
```

Example: card game

with separate compilation, .h file serves as quick index

```
// Card.h
////////////////////////////////////

#ifndef _CARD_H
#define _CARD_H

using namespace std;

const string SUITS = "SHDC";
const string RANKS = "23456789TJQKA";

class Card {
public:
    Card(char r = '?', char s = '?');
    char GetSuit() const;
    char GetRank() const;
    int GetValue() const;
private:
    char rank;
    char suit;
};

#endif
```

```
// Card.cpp
////////////////////////////////////

#include <iostream>
#include <string>
#include "Die.h"
#include "Card.h"
using namespace std;

Card::Card(char r, char s) {
    rank = r;
    suit = s;
}

char Card::GetRank() const {
    return rank;
}

char Card::GetSuit() const {
    return suit;
}

int Card::GetValue() const {
    for (int i = 0; i < RANKS.length(); i++) {
        if (rank == RANKS.at(i)) {
            return i+2;
        }
    }
    return -1;
}
```

Example: card game

templated vector class is
similar to Java's ArrayList

[] overloaded for vectors

```
// DeckOfCards.h
////////////////////////////////////
#ifndef _DECKOFCARDS_H
#define _DECKOFCARDS_H

#include <vector>
#include "Card.h"
using namespace std;

class DeckOfCards {
public:
    DeckOfCards();
    void Shuffle();
    Card DrawFromTop();
    bool IsEmpty() const;
private:
    vector<Card> cards;
};

#endif
```

```
// DeckOfCards.cpp
////////////////////////////////////
#include <string>
#include "Die.h"
#include "Card.h"
#include "DeckOfCards.h"
using namespace std;

DeckOfCards::DeckOfCards() {
    for (int suitNum = 0; suitNum < SUITS.length(); suitNum++) {
        for (int rankNum = 0; rankNum < RANKS.length(); rankNum++) {
            Card card(RANKS.at(rankNum), SUITS.at(suitNum));
            cards.push_back(card);
        }
    }
}

void DeckOfCards::Shuffle() {
    Die shuffleDie(cards.size());

    for (int i = 0; i < cards.size(); i++) {
        int randPos = shuffleDie.Roll()-1;
        Card temp = cards[i];
        cards[i] = cards[randPos];
        cards[randPos] = temp;
    }
}

Card DeckOfCards::DrawFromTop() {
    Card top = cards.back();
    cards.pop_back();
    return top;
}

bool DeckOfCards::IsEmpty() const {
    return (cards.size() == 0);
}
```

13

Example: card game

main is a stand-
alone function,
automatically called if
present in the file

(similar to public
static void
main in Java class)

```
#include <iostream>
#include <string>
#include "Card.h"
#include "DeckOfCards.h"
using namespace std;

int main() {
    DeckOfCards deck1, deck2;

    deck1.Shuffle();
    deck2.Shuffle();

    int player1 = 0, player2 = 0;
    while (!deck1.IsEmpty()) {
        Card card1 = deck1.DrawFromTop();
        Card card2 = deck2.DrawFromTop();

        cout << card1.GetRank() << card1.GetSuit() << " vs. "
             << card2.GetRank() << card2.GetSuit();

        if (card1.GetValue() > card2.GetValue()) {
            cout << ": Player 1 wins" << endl;
            player1++;
        }
        else if (card2.GetValue() > card1.GetValue()) {
            cout << ": Player 2 wins" << endl;
            player2++;
        }
        else {
            cout << ": Nobody wins" << endl;
        }
    }
    cout << endl << "Player 1: " << player1
         << " Player2: " << player2 << endl;

    return 0;
}
```

14

Other examples

Boggle

- utilizes 2-D array to store the board
- performs recursive backtracking to search the board

Bank Simulation

- utilizes multiple classes, Die for randomly distributed customer arrivals
- queue and vector classes for managing customers

Spell Checker

- utilizes BinarySearchTree class to store the dictionary
- reads and processes files

15

Java

Java was developed at Sun Microsystems, 1995

- originally designed for small, embedded systems in electronic appliances
- initial attempts used C++, but frustration at limitations/pitfalls

recall: C++ = C + OOP features

the desire for backward compatibility led to the retention of many bad features

desired features (from the Java white paper):

| | | |
|----------------------|-----------------|------------------|
| simple | object-oriented | network-savvy |
| interpreted | robust | secure |
| architecture-neutral | portable | high-performance |
| multi-threaded | dynamic | |

note: these are desirable features for any modern language (+ FREE)

→ Java has become very popular, especially when Internet related

16

Language features

simple

- syntax is based on C++ (familiarity → easier transition for programmers)
- removed many rarely-used, confusing features
e.g., operator overloading, multiple inheritance, automatic coercions
- added memory management (reference count/garbage collection hybrid)

object-oriented

- OOP facilities similar C++, but all member functions (methods) dynamically bound
- pure OOP – everything is a class, no independent functions*

network-savvy

- extensive libraries for coping with TCP/IP protocols like HTTP & FTP
- Java applications can access remote URL's the same as local files

17

Language features (cont.)

robust

- for embedded systems, reliability is essential
- Java combines extensive static checking with dynamic checking
 - closes C-style syntax loopholes
 - compile-time checking more effective
 - even so, the linker understands the type system & repeats many checks
- Java disallows pointers as memory accessors
 - arrays & strings are ADTs, no direct memory access
 - eliminates many headaches, potential problems

secure

- in a networked/distributed environment, security is essential
- execution model enables virus-free*, tamper-free* systems
 - downloaded applets cannot open, read, or write local files
- uses authentication techniques based on public-key encryption

note: the lack of pointers closes many security loopholes by itself

18

Language features (cont.)

architecture-neutral

- want to be able to run Java code on multiple platforms
- neutrality is achieved by mixing compilation & interpretation
 1. Java programs are translated into *byte code* by a Java compiler
 - byte code is a generic machine code
 2. byte code is then executed by an interpreter (Java Virtual Machine)
 - must have a byte code interpreter for each hardware platform
 - byte code will run on any version of the Java Virtual Machine
- alternative execution model:
 - can define and compile applets (little applications)
 - not stand-alone, downloaded & executed by a Web browser

portable

- architecture neutral + no implementation dependent features
 - size of primitive data types are set
 - libraries define portable interfaces

19

Language features (cont.)

interpreted

- interpreted → faster code-test-debug cycle
- on-demand linking (if class/library is not needed, won't be linked)

does interpreted mean slow?

high-performance

- faster than traditional interpretation since byte code is "close" to native code
- still somewhat slower than a compiled language (e.g., C++)

multi-threaded

- a *thread* is like a separate program, executing concurrently
- can write Java programs that deal with many tasks at once by defining multiple threads (same shared memory, but semi-independent execution)
- threads are important for multi-media, Web applications

20

Language features (cont.)

dynamic

- Java was designed to adapt to an evolving environment

e.g., the fragile class problem

in C++, if you modify a parent class, you must recompile all derived classes

in Java, memory layout decisions are NOT made by the compiler

- instead of compiling references down to actual addresses, the Java compiler passes symbolic reference info to the *byte code verifier* and the *interpreter*
- the Java interpreter performs name resolution when classes are being linked, then rewrites as an address
- thus, the data/methods of the parent class are not determined until the linker loads the parent class code
- if the parent class has been recompiled, the linker automatically gets the updated version

Note: the extra name resolution step is price for handling the fragile class problem