

Introducing C++ to Java Programmers

by Kip Irvine
updated 2/27/2003

1

Philosophy of C++

- Bjarne Stroustrup invented C++ in the early 1980's at Bell Laboratories. First called "C with classes".
- Design Goals:
 - Compatibility with existing ANSI C code (C++ compilers must compile ANSI C programs)
 - Object-oriented capabilities
 - Efficient running speed (native code, little or no runtime checks)
 - Simple language syntax (semantics are complex!)
- Constantly Evolving
 - compiler vendors are always behind the latest standard
 - example: templates and namespaces came later

2

Compatibility

- C++ standard is intentionally fuzzy on many implementation details
- For a long time, C++ was not standardized, so compiler vendors interpreted the specification in a variety of ways.
 - examples: string class, container library, input/output formatting
 - be suspicious of any C++ compiler released before 1997-1998

3

Important C++ Libraries

- Standard C Library
 - low-level functions for manipulating strings, arrays, input-output, and math. Not object-oriented.
- Standard C++ Library
 - containers and algorithms
 - incorporates many classes from the Standard C library.
- STL (Standard Template Library)
 - Containers and Algorithms

4

C++'s Lax Security

- C++ allows unsecure or unsafe code to compile and run. Examples:
 - uninitialized variables can be used
 - programmers must do their own garbage collection
 - memory leaks can result
 - dangling pointer can crash a program
 - no index checking on arrays
 - might crash, or might corrupt other data
 - buffer overflow (common hacker attack)
 - invalid typecasts

5

Multithreading

- Not part of C++ language
- Supported by specific compiler vendor libraries
 - example: Microsoft Foundation Classes (MFC)

6

But C++ is Really OK

- Don't let the previous slides scare you
 - just have to learn to be careful
- C++ is a great language, with a huge installed code base
 - most of the world uses C++ for high-performance applications
- You can get paid a lot of money if you're good
- Might have to learn the Windows API
 - (spend another year doing that)
- Alternative: managed code in Microsoft C++ .Net

7

Defining Data

Last Update: 11/4/2003

Copyright Kip Irvine, 2003

8

Const Qualifier

- The const qualifier prevents an object from being modified after its initial definition.
- A const-qualified object must be given a value when it is defined.
- A const-qualified object can be assigned to a non-const object.

```
const int n = 25;
n = 36;           // error
const double z;  // error
int m = n;
m = 36;
```

9

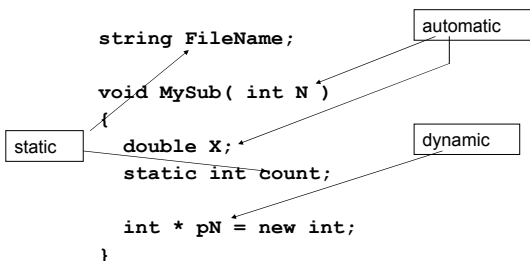
Storage Allocation

- A variable with static storage allocation exists for the lifetime of the program.
 - Global variables
 - Variables declared with static qualifier
- A variable with automatic storage is created on the stack when declared inside a code block.
 - Local function variables
 - Function parameters
- A variable with dynamic storage allocation exists on the heap, and may be both created and removed at runtime (*new* and *delete* operators).

10

Storage Allocation Examples

```
string FileName;
void MySub( int N )
{
    double X;
    static int count;
    int * pN = new int;
}
```



11

Variable Scope

- A global variable exists outside of any code block.
 - a block is described by the { ... } symbols
- A local variable exists inside a code block.
 - Inside a function, for example
- A member variable exists inside a class definition block
 - class *name* { ... }

12

References

- A reference is an *alias* for some existing object.
- Physically, the reference stores the address of the object it references.
- In the following example, when we assign a value to rN, we automatically modify N:

```
int N = 25;
int & rN = N;
rN = 36;
cout << N;           // "36" displayed
```

13

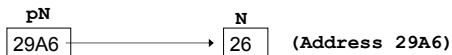
Pointers

- A pointer stores the address of some other object.
- The *address-of* (&) operator obtains the address of an object.

```
int N = 26;
int * pN = &N;    // get the address
```

14

Implementing a Pointer



pN points to N because pN contains N's address

15

Pointer Variables

A pointer variable must be declared with the same type it points to. In the following, pN cannot point to Z because Z is a double:

```
double Z = 26;
int * pN;
pN = &Z;           // error!
```

The internal format and size of a double is not the same as an integer!

16

Dereference Operator

The dereference (*) operator obtains the contents of the variable that is referenced by a pointer.

```
int N = 26;
int * pN = &N;
cout << *pN << endl;    // "26"
```

Only pointers can be dereferenced.

17

Dereference Operator

The dereference operator can also be used to modify the contents of the referenced variable.

```
int N = 26;
int * pN = &N;

*pN = 35;
cout << *pN << endl;    // "35"
cout << N << endl;      // "35"
```

Assigning a value to *pN changes the value of N.

18

Pointers and Dynamic Memory Allocation

Revised 10/28/2003

Copyright Kip Irvine 2003, all rights reserved.

19

Pointer Variables

A pointer variable is able to contain the address of some other variable. The address-of operator (&) gets the address of a variable.

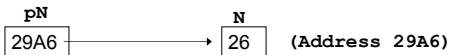
```
int N = 26;

int * pN;    // declare a pointer
pN = &N;    // assign the address of N
```

int * is a data type called pointer to integer.

20

Implementing a Pointer



pN points to N because pN contains N's address

21

Pointer Variables

Pointer variables are strongly typed. In the following example, pN cannot point to Z because Z is a double:

```
double Z = 26;

int * pN;
pN = &Z;    // compiler error
```

22

Dereference Operator

The dereference operator (*) obtains the contents of the variable that is referenced by a pointer.

```
int N = 26;
int * pN = &N;

cout << *pN << endl;    // "26"
```

23

Dereference Operator

The dereference operator can also be used to modify the contents of the referenced variable.

```
int N = 26;
int * pN = &N;

*pN = 35;
cout << *pN << endl;    // "35"
cout << N << endl;    // "35"
```

Assigning a value to *pN changes the value of N.

24

Assigning Pointers

One pointer may be assigned to another, as long as they point to the same type. In this example, when pZ is dereferenced, it lets us change the value of N:

```
int N = 26;
int * pN = &N;
int * pZ;

pZ = pN;
*pZ = 35;           // now N = 35
```

25

Assigning Pointers

Assigning a value from one pointer to another can be done by dereferencing both pointers.

```
int N = 26;
int * pN = &N;
int Z = 0;
int * pZ = &Z;

*pZ = *pN;           // Z = 26
```

26

Other Pointers

You can create pointers to any data type. A pointer to an object is dereferenced with the `->` operator when calling a member function.

```
Student aStudent( "100-33-3333", "Johnson" );
Student * pS = &aStudent;

cout << pS->getIDNumber();
cout << (*pS).getIDNumber();
```

↑
Alternate form

27

Uninitialized Pointers

Beware of dereferencing a pointer before it has been initialized. This causes a runtime error (*invalid pointer reference*).

```
int n = 30;
int * p;

*p = n;           // runtime error
```

28

Uninitialized Pointers (cont)

Similar example using an object pointer:

```
string * pS;

int n = pS->length;           // error
```

29

Initializing a Pointer

NULL is the best default value to assign to a pointer if you cannot assign it the address of an object. A smart programmer will check for NULL before using the pointer.

```
int * pN = NULL;
.
.
// ...later,

if( pN != NULL )
    *pN = 35;
```

30

Dynamic Allocation

- Use dynamic allocation to create an object at runtime. C++ uses the new operator.
- The object is stored in a large free memory area named the heap (or *free store*).
- The object remains on the heap either until you remove it or the program ends.
- The delete operator erases an object from the heap.

31

Creating an Object

Create an int object on the heap and assign its address to P:

```
int * P = new int;
```

Use the pointer in the same way as previous examples:

```
*P = 25; // assign a value
cout << *P << endl;
```

32

new and delete

The new operator returns the address of a new object. The delete operator erases the object and makes it unavailable.

```
Student * pS = new Student;
.
.
// use the student for a while...
.
.
delete pS; // gone!
```

Student constructor called

33

Using new in Functions

If you create an object inside a function, you may have to delete the object inside the same function. In this example, variable pS goes out of scope at the end of the function block.

```
void MySub()
{
    Student * pS = new Student;
    // use the Student for a while...

    delete pS; // delete the Student
              // pS disappears
}
```

34

Memory Leak

A memory leak is an error condition that is created when an object is left on the heap with no pointer variable containing its address. This might happen if the object's pointer goes out of scope:

```
void MySub()
{
    Student * pS = new Student;
    // use the Student for a while...
} // pS goes out of scope

(the Student's still left on the heap)
```

35

Function Returning an Address

A function can return the address of an object that was created on the heap. In this example, the function's return type is *pointer to Student*.

```
Student * MakeStudent()
{
    Student * pS = new Student;
    return pS;
}
```

(more) →

36

Receiving a Pointer

(continued)...

The caller of the function can receive the address and store it in a pointer variable. As long as the pointer remains active, the Student object is accessible.

```
Student * pS;

pS = MakeStudent();

// now pS points to a Student
```

37

Dangling Pointer

A dangling pointer is created when you delete its storage and then try to use the pointer. It no longer points to valid storage and may corrupt the program's data.

```
double * pD = new double;
*pD = 3.523;
.
.
delete pD; // pD is dangling...
.
.
*pD = 4.2; // error!
```

38

Avoid Dangling Pointers

To avoid using a dangling pointer, assign NULL to a pointer immediately after it is deleted.

And, of course, check for NULL before using the pointer.

```
delete pD;
pD = NULL;
.
.
if( pD != NULL ) // check it first...
    *pD = 4.2;
```

39

Passing Pointers to Functions

Passing a pointer to a function is almost identical to passing by reference. The function has read/write access to the data referenced by the pointer.

This is how swap() was written in C, before C++ introduced reference parameters:

```
void swap( int * A, int * B )
{
    int temp = *A;
    *A = *B;
    *B = temp;
}
```

40

Const-Qualified Pointer

A const-qualified pointer guarantees that the program has read-only access to the data referenced by the pointer.

```
void MySub( const int * A )
{
    *A = 50; // error
    A++; // ok
}
```

The pointer itself can be modified, but this has no lasting effect--the pointer is passed by value.

41

Constant Pointer

Declaring a constant pointer guarantees only that the pointer itself cannot be modified.

```
void MySub( int * const A )
{
    *A = 50; // ok
    A++; // error
}
```

The data referenced by the pointer can still be modified.

42

Arrays and Pointers

An array name is assignment-compatible with a pointer to the array's first element. In the following example, *p refers to scores[0].

```
int scores[50];
int * p = scores;
*p = 99;
cout << scores[0];    // "99"

p++;                // ok
scores++;           // error: scores is const
```

43

Traversing an Array

C and C++ programmers often use pointers to traverse arrays. At one time, such code ran more efficiently, but recent optimizing compilers make array subscripts just as efficient.

```
int scores[50];
int * p = scores;

for( int i = 0; i < 50; i++)
{
    cout << *p << endl;
    p++;                // increment the pointer
}
```

44

Array of Pointers

An array of pointers usually contains the addresses of dynamic data objects. This keeps the storage used by the array itself quite small, and puts most of the data on the heap.

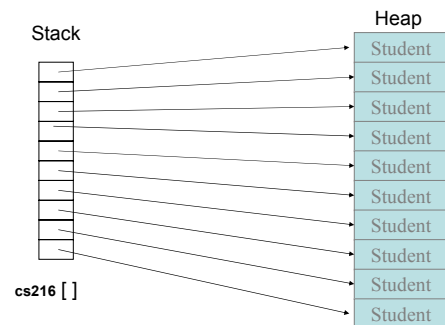
```
Student * cs216[10];

for(int i = 0; i < 10; i++)
{
    cs216[i] = new Student;
}
```

diagram →

45

Array of Pointers



46

Creating an Array on the Heap

You can create an entire array on the heap, using the new operator. Just remember to delete it before the program exits. Include "[]" before the array name in the delete statement.

```
void main()
{
    double * samples = new double[10];
    // samples is now an array...
    samples[0] = 36.2;

    delete [] samples;
}
```

47

Pointers in Classes

Pointers are effective when encapsulated in classes, because you can control the pointers' lifetimes.

```
class Student {
public:
    Student();

    ~Student();

private:
    string * courses; // array of course names
    int count;        // number of courses
};

// more...
```

48

Pointers in Classes

The constructor creates the array, and the destructor deletes it. Very little can go wrong,...

```
Student::Student()
{
    courses = new string[50];
    count = 0;
}

Student::~Student()
{
    delete [] courses;
}
```

49

Pointers in Classes

...except when making a copy of a Student object. The **default** copy constructor used by C++ leads to problems. In the following example, a course assigned to student X ends up in the list of courses for student Y.

```
Student X;
Student Y(X);           // construct a copy

X.AddCourse("cs 216");

cout << Y.GetCourse(0); // "cs 216"
```

50

Pointers in Classes

To prevent this sort of problem, we create a copy constructor that performs a so-called *deep copy* of the array from one student to another.

```
Student::Student(const Student & S2)
{
    count = S2.count;
    courses = new string[count];

    for(int i = 0; i < count; i++)
        courses[i] = S2.courses[i];
}
```

51

Pointers in Classes

For the same reason, we have to overload the assignment operator.

```
Student & Student::operator=(const Student & S2)
{
    delete [] courses; // delete existing array
    count = S2.count;

    for(int i = 0; i < count; i++)
        courses[i] = S2.courses[i];

    return *this;
}
```

52

STL Containers in Classes

When you use lists and vectors in classes, there is no problem with the default copy constructor in C++. All STL containers are experts at allocating and copying themselves.

```
class Student {
public:
    Student();

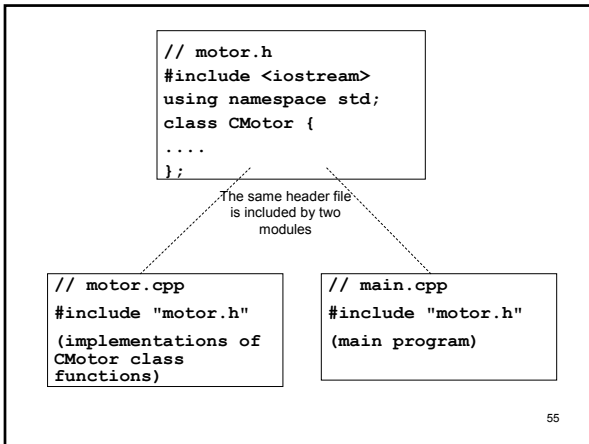
private:
    vector<string> courses;
};
```

53

Modules

Copyright Kip Irvine 2003, all rights reserved

54



Header File

- Each class is defined in a header file having a similar name to the class
 - CMotor = motor.h
 - Student = student.h
 - Point = point.h
- The header file should be added to the Visual C++ Workspace

56

Defining Symbols

- C++ permits **preprocessor macros** to define symbols and check for existing symbol definitions.
- Define a new symbol:


```
#define MY_SYMBOL
```
- Enable a block of code if a symbol has been defined:


```
#ifdef MY_SYMBOL
    code...
    code...
#endif
```

57

motor.h

```

#ifndef _MOTOR_H
#define _MOTOR_H

#include <iostream>
#include <fstream>
#include <string>
using namespace std;

class CMotor {
    ...
};

#endif

```

These statements avoid including the header twice

Standard C++ headers

End of conditional text

58

Implementation File

- Each class is implemented in a CPP file having a similar name to the class
 - CMotor = motor.cpp
 - Student = student.cpp
 - Point = point.cpp
- The CPP file must be added to the Visual C++ workspace

59