EECS 280

Effective C++

Mark C. Davis

Fall 1998

1. Use C++ Standard I/O

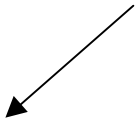
```
int i;
ComplexInt c;

scanf("%d ?", I);       // C scanf is not extensible
printf("I ?",I,c);      // C printf is not extensible
                -or-
cin >> i >> c;          // C++ call to operator >>
cout << i << c;         // call operator<<
```

Why?

- type safety
- scanf/prinf not extensible
- worry about & in scanf

2. Use C++ Memory Allocation

```
class String {
public:
  String(const char *val = 0);
  ~String();
};
```



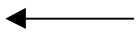
- no constructor call
- no initialization

```
String *StringArray1 = (String *) malloc(10*sizeof(String));
```

```
String *StringArray2 = new String[10];
```

- calls the constructor
- gets initialization

```
free (StringArray1);
```



- no destructor call

```
delete [] StringArray2;
```

- call to the destructor

combinations are undefined in the ANSI standard

3. Use C++ Comments

- /* */ cannot be nested

```
If (a > b) {
 /* int temp = a;
     a = b;
     b = temp;
*/
```

```
/* swap a and b */
```

this terminates comment

- use /* */ for header files used for both C and C++

4. Avoid #define Misuse

const int PIE 3.14

or

#define PIE 3.14

#define MAX(a,b) ((a) > (b) ? (a) : (b))

int a=1, b=0;
MAX(a++, b); // a increments twice
MAX(a++,b+10); // a increments once
MAX(a, "hello"); // comparing ints to pointers

```
inline int MAX(int a, int b)
{
   return a>b ? a : b;
}
```

- Efficiency of a macro
- Type safety

```
template<class T>
inline T& MAX(T& a, T& b)
{
   return a>b ? a : b;
}
```

- Handles all types
- Pass by reference
  - efficient: no copy constructor called
  - "slicing problem"
- Compiler MAGIC

5. Use <assert.h>

Memory allocation:

```
#include <assert.h>
. . .
lastName = new char[ strlen( last ) + 1 ];
assert( lastName != 0 );                    // ensure memory allocated
```

Assert() functionality:

- value True
  - nothing happens
- value False
  - prints line #
  - calls abort() from stdlib.h

6. Check the Return Value for New

```
#include<assert.h>
#define NEW(PTR, TYPE) (PTR) = new TYPE; assert((PTR) != 0)
```

What about?

```
new T;
new T(args);
new T[size];
```

The global case

```
#include<new.h>
extern void (*set_new_handler (void (*) ()))   ();
void noMoreMemory() {
   cerr << “Unable to satisfy request for memory\n”;
   abort();
}
main() {
   set_new_handler(noMoreMemory);
   char *bigString = new char[100000000];
   . . .
}
```

7. Prefer Initialization to Assignment

```
class NamedData {
private:
  String name;
  void *data;

public:
  NamedData(const String& initName, void *dataPtr);
};
```

Method 1 is better
- const and ref members require it
- efficiency
  - only calls copy constructor

Method 2 is worse
- calls the default constructor
- then, calls the operator=
- good for consistent initialization

Method 1:

```
NamedData::NamedData(const String& initName, void *dataPtr) : name(initName), data(dataPtr) { }
```

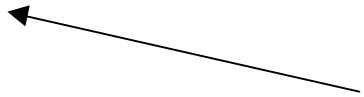
Method 2:

```
NamedData::NamedData(const String& initName, void *dataPtr) {
  name = initName;
  data = dataPtr;
}
```

8. Declaration == Initialization Order

```
class Array {
private:
   int *data;          <-- #1
   unsigned size;      <-- #2
   int lBound, hBound; <-- #3, #4

public:
   Array(int lowBound, HighBound);
};
```

```
Array::Array(int lowBound, int highBound) : size(highbound - lowbound + 1), lbound(lowBound),
  hBound(highBound), data(new int[size]) {}
```

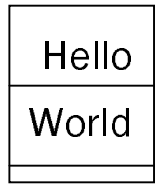


- size is Undefined!!!

9. When to explicitly define Operator=

```
class String {
private:
  char *data;
public:
  String(const char *value = 0);
  ~String();
};

String::String(const char *value) {
  if (value) {
    data = new char[strlen(value) + 1];
    strcpy(data, value);
  } else {
    data = new char[1];
    *data = '\0';
  }
}

inline String::~String() { delete [] data; }
```

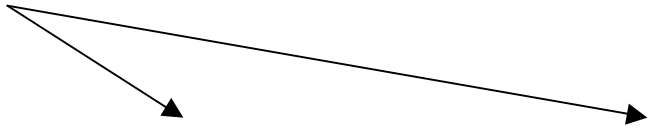
```
Bad_Main1 {
  String a("Hello");
  String b("World");

  b = a;     // calls default operator
             // memberwise copy of data
             // bitwise copy on a.data and b.data
             // !!! both a and b point to Hello
             // b is never deleted
             // a destructor kills a and b
}

Bad_Main2() {
String a("Hello");
{
  String b("World");
  b = a;
}                          // b destructor kills a and b
String c = a               // undefined!
}
```



10. Explicitly define Copy Constructor

```
Void doNothing(String localstring) {};

main() {
  String s = "Goodbye cruel world";
  doNothing(s);                    // kills s when the call the destructor is made
  delete s;                        // undefined!!
}
```

If you see **new** anywhere in the class



explicitly define the assignment operator= and the copy constructor

11. Usually Pass by Reference

```
Student returnStudent(Student p) { return p; }

main() {
    Student s;
    returnStudent(s);
}
```

1. Call copy constructor to init p with s
2. Call copy constructor to init return object
3. Destructor called for p
4. Destructor called for object returned

But, there is more.

5. Each Student contains two String objects
6. Each Student inherits from Person
7. Each Person object has two String objects

```
class String { ... };

class Person {
private:
    String name, address;
public:
    Person();
    ~Person();
    . . .
};

class Student: public Person {
private:
    String schoolName, schoolAddress;
public:
    Student();
    ~Student();
    . . .
};
```

= 1 Student copy + 1 Person copy + 4 String copy x 2 (destructor calls) = **12 calls**

11. (cont.) Pass by Reference: Slicing Prob

```
class Window {
public:
  const char * name() const;     // return name of window
  virtual void display() const;  // draw window and contents
};

class WindowWithScrollBars: public Window {
public:
  virtual void display() const;
};

// a function that suffers from the slicing problem
void printNameAndDisplay(Window w) {
  cout << w.name();
  w.display();
}
void printNameAndDisplayCorrect(const Window& w) {
  cout << w.name();
  w.display();
}
```

By value

By reference

```
Main() {
  WindowWithScrollBars wwsb;
  printNameAndDisplay(wwsb);
}
```

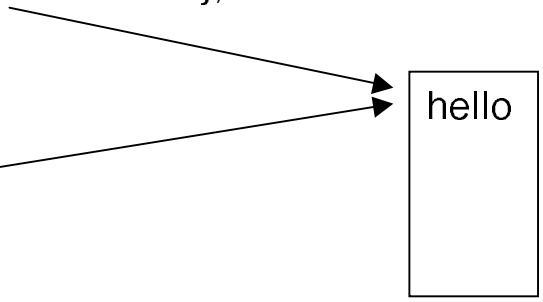
1. w constructed as Window object
2. All special information of wwsb is lost
3. w.display calls Window member fcn instead of WindowWithScrollBars::

Correct call made for w.display()

12. Check Assignment to Self: Aliasing Prob

```
class X { . . . };

X a;
a = a;      // obvious case
a = b;      // b is ref. Initialized to a

// assignment operator with check to self
String& String::operator=(const String& rhs)
{
  delete [] data;
  data = new char[strlen(rhs.data) + 1];
  strcpy(data, rhs.data);
  return *this;
}

main() {
  String a = "Hello";
  a = a;    // a.operator=(a)
}
```

```
Class String {
private:
  char *data;
public:
  String(const char *value = 0);
  ~String();
  String& operator=(const String& rhs);
};
```



1. delete call deleted hello
2. Strlen value (rhs.data) is undefined

Check for assignment to self

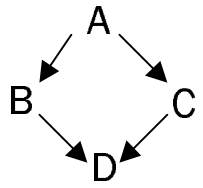
- efficiency
- correctness

12. (cont.) More on Aliasing: Object Identity

The solution to the aliasing problem is to check for assignment to self.

- What does it mean for an object to be "the same?"
  - same value?
    - Check: if (strcmp(data, rhs.data) == 0) return *this;
    - Check: if (*this == rhs) return *this; // assumes ==
  - same address? (more common in C++)
    - Check: if (this == &rhs) return *this;
    - only reliable for single inheritance
      - a single object can have more than one address, with or without virtual fcns

```
class A {. . .};
class B: public A { ... };
class C: public A { ... };
class D: public B, public C { ... };
```




```
D d;
D* pD1 = &d;        // address of D part of D
B* pD2 = &d;        // address of B part of D
C* pD3 = &d;        // address of C part of D
A* pD4 = (B*) &d;   // address of A part of B
A* pD5 = (C*) &d;   // address of A part of C
```

- There is no guarantee that any of these pointers will have the same value

12. (cont.) Solving Object Identity

```
Class C {
public:
  virtual ObjectID identity() const;
   ...
};
```

* don't forget cases w/pointers instead of references

```
// the test
if (a->identity() == b->identity())
```

```
// must overload == for ObjectID class
```

```
// example 1
class Base {
   void mf1(Base& rb);
   // rb and *this must be checked
};
```

```
// example 2
void f1(Base& rb1, Base& rb2);
// rb1 and rb2 could be the same
```

```
// example 3
class Derived: public Base {
   void mf2(Base& rb);
   // rb and *this must be checked
};
```

```
// example4
int f2(Derived& rd, Base& rb);
// rd and rb could be the same
```

13. Operator= return *this

With Builtin Types:

```
Int w,x,y,z;
w = x = y = z = 0;
```

= is right associative. So,
w.operator=(x.operator (y.operator=(z.operator=("Hello"))));

The Standard Assignment Operator:
C& C::operator=(const C&);

User-defined Types:

```
Class String {
private:
   char *data;
public:
   String(const char *value = 0);
};

String w,x,y,z;
w = x = y = z = "Hello";
```

String has two of them:
String& operator=(const String&);
String& operator=(const char*);

```
String& String::operator=(const String& rhs) {
  . . .
  return *this;        // returns lhs
    - or -
  return rhs;          // returns rhs, remove const
}
```

x = "Hello"; —> String temp("Hello"); x = temp;
Problem: How long is temp available? Who knows...

14. Update operator=

```
class Point {
private:
   long x,y;
public:
   Point(long xCoord, long yCoord) : x(xCoord), y(yCoord) { }
   Point& operator=(const Point& rhs);
   ...
};

Point& Point::operator=(const Point& rhs) {
   if (this == &rhs) return *this;                    // see #12
   x = rhs.x;
   y = rhs.y;
   return *this;                                      // see #13
}
```

- Remember to update the assignment operator when you add to a class!

15. Inheritance and operator=

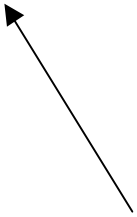
```
// Hosed assignment operator
B& B::operator=(const B& rhs) {
    if (this == &rhs) return *this;
    y = rhs.y;
    return *this;
}

Main() {
    B b0(0);  ←  B0.x = 0, b0.y = 0
    B b1(1);  ←  b1.x = 1, b1.y = 1
    b0 = b1;  ←  b0.x = 0, b0.y = 1!!
}

// Correct assignment operator
B& B::operator=(const B& rhs) {
    if (this == &rhs) return *this;
    ((A&) *this) = rhs;  ←  Same as A::operator=(rhs);
    y = rhs.y;
    return *this;
}
```

```
class A {
private:
    int x;
public:
    A(int initialValue): x(initialValue) { }
};

class B : public A {
private:
    int y;
public:
    B(int initialValue): A(initialValue), y(initialValue) { }
    B& operator=(const B& rhs);
};
```

16. Make Base Class Destructors Virtual

```
class EnemyTarget {
private:
   static unsigned numTargets;
public:
   EnemyTarget() { numTargets++; }
   EnemyTarget(const EnemyTarget&) { numTargets++; }
   ~EnemyTarget() { numTargets--; }

   static unsigned numberOfTargets() { return numTargets; }
   virtual boolean destroy();
};
unsigned EnemyTarget::numTargets;

class EnemyTank: public EnemyTarget {
private:
   static unsigned numTanks;
public:
   EnemyTank() { numTanks++; }
   EnemyTank(const EnemyTank&) { numTanks++; }
   ~EnemyTank() { numTanks--; }
   static unsigned numberOfTanks() { return numTanks; }
   virtual boolean destroy();
};
```

```
Main() {

EnemyTarget *targetPtr = new EnemyTank;
. . .
delete targetPtr;
}
```



- numTanks is now **incorrect!!!**
- Calls **EnemyTarget** destructor

17. Only Base Class Destructors Virtual

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

No Virtual Functions
Point Size: 32-bits (2) 16-bit ints

Virtual Functions
Point Size: 32-bits + 32-bit vptr

- If (int) is 16-bits, a Point object fits into a 32-bit reg
- this is not true if the destructor is made virtual!
- Vptr points to vtbl (per class), which is an array of function ptrs
- this class will no longer fit into a 32-bit register
- if you must, inline the destructor call. the compiler may ignore you anyway.

Declare a virtual destructor for functions with one or more virtual functions.

18. Return an Object when necessary

"Make things as simple as possible, but no simpler" Albert Einstein

```
class Complex {
private:
   double r, I;
public:
   Complex(double realPart = 0, double imagPart = 0);
   ~Complex();
   friend Complex operator+(const Complex& lhs, const Complex& rhs);
);

inline Complex operator+(const Complex& lhs, const Complex& rhs) {
   return Complex(lhs.r + rhs.r, lhs.I + rhs.I);
}

main() {
   Complex a(3,2);
   Complex b(-5, 22);
   Complex c = a+b;
}
```

- Can we avoid the constructor call?
- Is it reasonable to expect c to already exist?

18. (cont.) When you shoudn't return a Reference

```
// first hosed case
// returning a reference to an object from the stack
inline Complex& operator+(const Complex& lhs, const Complex& rhs) {
   Complex result(lhs.r + rhs.r, lhs.I + rhs.I);
   return result;
}
```

- Calls the constructor, bad idea
- and...it returns a ref to a local!

```
// second bad implementation
// returning a reference to an object on the heap
inline Complex& operator+(const Complex& lhs, const Complex& rhs) {
   Complex *result = new Complex(lhs.r + rhs.r, lhs.I + rhs.I);
   return *result;
}
```

- Still calls constructor w/new
- How will it get deleted?

Consider w = x + y +z;
// unnamed temps !

19. Use const whenever you can

- Outside classes
  - Global constants
  - static objects (file/block scope)
- Inside classes
  - static/non-static data members
- Pointers

	char *		p = “Hello”;
const	char *		p = “Hello”;
	char *	const	p = “Hello”;
const	char *	const	p = “Hello”;

What is pointed to is constant

Pointer itself is constant

20. Fcn. Overloading vs. Parameter Defaulting

```
void f();
void f(int x);

f();                    // calls f()
f(10);                  // calls f(int)

void g(int x = 0);

g();                    // calls g(0)
g(10);                  // calls g(10)
```

```
// parameter defaulting is good this type of situation
int max(int a, int b = INT_MIN, int c = INT_MIN,
        int d = INT_MIN, int e = INT_MIN) {
   int temp = a > b ? A : b;
   temp = temp > c ? temp : c;
   temp = temp > d ? temp : d;
   temp = temp > e ? temp : e;
}


// have to overload here
int avg(int a);
int avg(int a, int b);
int avg(int a, int b, int c);
int avg(int a, int b, int c, int d);
int avg(int a, int b, int c, int d, int e);
```

21. Do not Overload Numerical and Ptr Types

```
void f(int x);
void f(char *p);

f(0);                    // what does this call?


// brute force solution
const int * const NULLint = 0;
const char * const NULLchar = 0;

// however, how can you guarantee the proper calls?
```

22. Explicitly disallow Implicit Functions

```
char string1[10];
char string2[10];

string1 = string2;      // not allowed in C++


Class Array {
private:
   Array& operator=(const Array& rhs);
   . . .
};
```

What if a friend or member tries to makes the call?

- Don't define the function !

23. Guard against Ambiguity

Case 1

```
class B;

class A {
public:
   A(class B&);
};

class B {
public:
   operator A() const;
};

Main() {
void g(const A&);
B b;        // note, this error will not show up!
g(b);       // how does compiler come up with
}           // object of type A?
            // call A constructor with B?
            // call client-defined conversion operator?
```

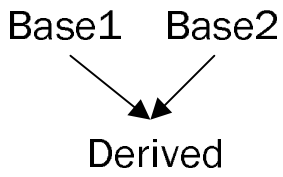
Case 2

```
Void h(int);
void h(char);

double pi = 3.14;
h(pi);          // ambiguous
                // convert to int?
                // convert to char?

H((int)pi);
```

23. (cont.) Guard against Ambiguity

Case 3

```
class Base1 {
public:
   int doIt();
};
class Base2 {
public:
   void doIt();
};
class Derived:public Base1, public Base2 {
};
Derived d;
d.doIt();                  // ambiguous, error

d.Base1::doIt();
d.Base2::doIt();
```



Case 4

```
class Base1 {
public:
   int doIt();
};
class Base2 {
private:
   void doIt();
};
class Derived:public Base1, public Base2 {
};
Derived d;
int i = d.doIt();          // still ambiguous
                           // even with private
d.Base1::doIt();           // and int return
d.Base2::doIt();
```

24. Use inlining Judiciously

```
// header file
inline void f() { ... }
...

// source1 code
#include "header.h"
...

// source2 code
#include "header.h"
...
```

Assuming f() not inlined

- f() inside of the source1.o object file
- f() inside of the source2.o object file

Link problem?

- static assumption
- converted to file scope

Keep in mind:

- Inline small functions, since the entry/exit code may actually bloat the code
- inlining increases the size of the object code (memory problems)
- inlining can cause pathological paging behavior (thrashing)
- inlining is just a suggestion
- most compilers will not inline recursive functions
- most compilers will warn you if they fail to inline a function
- inlining can be bad when you suggest it and the compiler does not do it, because you suffer from code bloat and overhead penalty

24. (cont.) More on Function inlining

```
inline void f() { ... }

void (*pf) () = f;

main() {
    f();        // inline call
   pf();        // NOT inline call
}               // a static copy is made
                // so the ptr can point
                // to something.
```

- Most debuggers cannot cope with inline functions
- SOLN: Be very selective about function inlining
- 80 to 90/20 rule

25. Know what happens behind the scenes

The compiler will write:

- a public copy constructor
- a public assignment operator
- and a pair of public address-of operators
- a public default constructor if necessary

```
const Empty e1;        // constructor
Empty e2 = e1;         // copy
e2 = e1;               // assignment
Empty *pe2 = &e2;      // address-of
const Empty *pe1 = &e1;  // const
```

class Empty(); →

```
class Empty {
public:
   Empty();                                  // default constructor
   Empty(const Empty& rhs);                  // copy constructor
   Empty & operator=(const Empty& rhs);      // assignment operator
   Empty* operator&();
   const Empty* operator&() const;
```

```
inline Empty::Empty() { }
inline Empty * Empty::operator&() { return this; }
inline const Empty * Empty::operator&() const { return this; }
```

assignment operator/copy constructor -> memberwise copy of all non-static data

25. (cont.) Know what happens behind the scenes

```
class String {
public:
   String(const char *value = 0);
   String(const String& rhs);
   String& operator=(const String& rhs);
   ...
};

class NamedInt {
private:
   String nameValue;
   int intValue;
public:
   NamedInt(char *name, int value);
   NamedInt(const String& name, int value);
   ...//no copy constructor or assignment operator
};
```

```
Main() {
   NamedInt i("Smallest Prime Number", 2);
   NamedInt j = I;      // calls copy constructor
}
```

j.nameValue calls the copy constructor for String
j.intValue copies the bits from I.intValue

Last example:

```
class B {
public:
   ~B();
};
class D: public B {};
// D::~D is implicitly generated as well
```

26. Beware of function hiding with Inheritance

```
Class Base {
public:
  void f(int c);
};

class Derived: public Base {
public:
  void f(char *p);
};

Derived *pd = new Derived;
pd->f(10);              // error!!
```

The Problem:

- Derived::f hides Base::f (compiler wants char *)

The Solution:

- Put the following in the derived class:

```
Void f(int x) { Base::f(x);}
```

27. When to use Static

- Use for class wide data
- No *this
- Reference static data with <class>::<var>
- You can reference (public) static data without the class instantiation. Use ::<var name>