C++ Part II

Software Development Training

C++ Part 2

Notes

Developed By:

Neal Goldstein
Neal Goldstein Design
659 Tennyson Avenue
Palo Alto, CA 94301
(415) 327-4565
AppleLink D0771

Notes

Table of Topics

Section 4 Inheritance

Section 5 Apple Extensions

Section 6 Future Directions

Section 1

Introduction

Notes

The Goals of This Class

- To be able to demonstrate competency in C++ by being able to write programs that incorporate C++ features
- Use the object-oriented features of C++ appropriately
- Understand the relationship and interactions between language features

Notes

This Section's Goals

- Understand the C++ approach and the differences between C++ and C
- Examine what things are difficult in C++ and how some language features can help

Notes

Features, Features, Features

Dynamic objects Dynamic binding Polymorphism Declarations in blocks
Private inheritance Constructors Pointer to members
Constant functions Operator overloading Load/Dump
Public inheritance Data protection
C terms Single inheritance
Operator Overloading Classes Symbolic constants
Assignment overrides
Type-safe linkage Function prototypes Objects
Implicit type conversions Pure Virtual Functions User defined types
Destructors Memberwise initialization Friends inline functions
New comments Initialization constructors Member functions
Apple extensions Streams
Automatic typedefs Operator functions Function overloading
new and delete Argument type checking
Pass by reference Virtual base classes
User Defined Conversions Static objects Reference variables
Static members Reduced name spaces Multiple inheritance

Notes

All This in Two Days?

- No way!
- Extending the language
  - Operator functions
    - Overload built in functions
  - User-specified conversions
    - Conversion functions and constructors
  - Memberwise assignment and initialization
    - Initialization constructors
    - Assignment operator overload
- Inheritance
- Apple extensions

Notes

C++ on the Macintosh

- Uses AT&T CFront preprocessor and MPW C compiler
- Shares MPW C header files

```
#ifdef __cplusplus
extern "C" {
#endif
pascal void InitGraf(void *globalPtr)
    = 0xA86E;
#ifdef __cplusplus
}
#endif
```

Notes

Each copy of CFront is licensed by Apple from AT&T
There is also a native C++ compiler produced by Zortech that runs as an MPW tool

C++

- Some call it a collection of features masquerading as a language
- Reasons

  Books and class written and taught by people without real object-oriented development experience

  The C++ concept of classes goes beyond what people normally think of classes

Notes

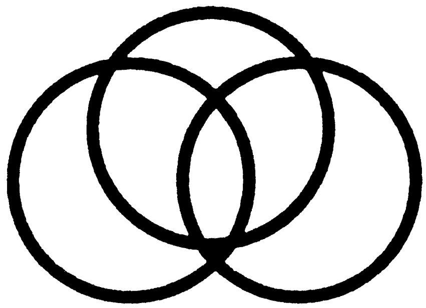
Design Goals of C++

- Support for object-oriented programming
- Support for data abstraction
- A better C
- Within the following constraints:
  - Compatibility with C
    - Requires C be a subset of C++
  - As efficient as C
    - C++ run time code performs as well
      - Implies no price for unused features

Notes

Alternative View

a. A better C
b. Object-oriented applications
c. Extending the language
d. Side effects of b. & c. that result in a.



Notes

Group Discussion

- Break up into groups of three
- Make a list of the most difficult things to do using C++

  What is (still) hard?

Notes

Misleading Error Messages

```
class TNamedObject {
public:
    void        AcceptName(char* aName);
...};

                              a
void  TNamedObject::AcceptNme(char* aName) {
    strcpy(fName, aName);}
```

error: AcceptNme() is not a member of TNamedObject

```
File "StudentIncM.cp"; line 100 # error: two initializers for TStudent
File "StudentIncM.cp"; line 100 # warning:  aAdvisor not used
File "StudentIncM.cp"; line 105 # error: two initializers for TFaculty
File "StudentIncM.cp"; line 105 # warning:  aAdvisee not used
File "StudentIncM.cp"; line 109 # error: two initializers for TStudent      ?
File "StudentIncM.cp"; line 109 # error: two initializers for TFaculty
File "StudentIncM.cp"; line 109 # warning:  aAdvisor not used
File "StudentIncM.cp"; line 109 # warning:  aAdvisee not used
File "StudentIncM.cp"; line 149 # error: two initializers for TStudent
```

Notes

Mysterious Unmangle Results

I forget to implement

```
class TGradStudent: public TStudent {...
    virtual    void  Warning() ; ... };

### link: Error: Undefined entry, name: (Error 28)
unmangle "Warning__11TGradStudentFv"
Unmangled symbol: TGradStudent::Warning()

class TStudent { ...
    virtual    void  Print(); ... };

### link: Error: Undefined entry, name: (Error 28)
unmangle "__ptbl__12TStudent"
Unmangled symbol: TStudent::__ptbl
```

Notes

Memory Allocation

Static

Class members

Stack

```
NewType aNewType;
```

Pointer-based dynamic

```
NewType* aNewType = new NewType;
```

Handle-based dynamic

Apple-only extension

```
NewTypeH* aNewTypeH = new NewTypeH;
```

Look the same (both `new` expressions above)

Notes

Creating Objects

String* stringHeap = new String("stringHeap"); ——Heap
stringHeap->Print();
(*stringHeap).Print();

String stringStack = String("stringStack"); ———— Stack
stringStack.Print();
(&stringStack)->Print();

Member selector operator depends on how message is sent

Pointer to object [->]

Object [•]

But you can only delete objects created with new

Notes

We create an object as defined by a class

- All instances of a class (object) share data structures and member functions

Class objects can be created on the stack or heap

- Local variables are instantiated (allocated and initialized)
- The class object is allocated enough storage for its data members [and pointer]

For static objects, the variable is the definition

- Stack space is allocated for all the data members

For dynamic objects, the variable is a pointer (or handle - Apple extension) to the object

- Objects are a non-relocatable (or relocatable - Apple extension) block on the heap

Arrays of class objects

```
TString* theStrings = new TString[somesize];
```

- If class has a constructor, it requires default constructor (constructor with no arguments)

Class or object

- People understand there is a difference between
  - The class of something
  - The thing itself (an instance)
- Daughter is a subclass of the girl-child class
  - But Sarah is my daughter

Function Overloading

TGradStudent();
TGradStudent(TFaculty* aAdvisor);
TGradStudent(TStudent* aAdvisee);
TGradStudent(TFaculty* aAdvisor, TStudent* aAdvisee);

Function signature

Number, order, and type of arguments

Notes

Two functions can have the same name as long as the types of their arguments differ i.e. their signatures are unique:

```
void MyPrint(char* s);
void MyPrint(int i);

main() {
 MyPrint("I love C++"); // MyPrint(char*) is invoked
 MyPrint(12);           // MyPrint(int) is invoked
}
```

Useful when you want to have different versions of the same function; they should all be related

```
Draw(EpsType);
Draw(PictType);
```

Function overloading rules

- If the return type and signatures match:
  - Redeclaration of the first
- If signatures match, but return types differ:
  - Erroneous redeclaration of the first
- If signatures differ in either number or type:
  - They are considered to be overloaded

When not to overload

- Functions do not perform similar operations

Rules and Resolution

- Functions are chosen by signature
- Application of standard and user-defined conversion
- Ambiguity

```
Move(int x);
Move (int x, int y=6);

...

Move(6);        ??????
```

Notes

Resolving the overloaded function call

Functions are chosen by signature through a process called argument matching

Compares actual arguments of the call with formal arguments of each declared instance

One of three results:

A match

No match

Ambiguous match

Matches

Exact match (trivial conversions allowed)

Match with promotions

Match with standard conversions

Match with conversions requiring temporaries

Match with user-defined conversions

Match with ellipsis

Can distinguish between const and ordinary pointer and reference

```
ff( const char*);
ff(char*);
```

Cannot distinguish between const and ordinary objects

```
ff(int);
ff(const int);  // makes no sense anyway (pass-by-value )
```

C Terms

Definition

```
int value1 = 1;
int& value1R = value1;
int value2 = 2;
```

	lvalue	rvalue
	value1 == value1R	1
	value2	2

```
value1 = value2;
```

	lvalue	rvalue
	value1 == value1R	2
	value2	2

Declaration

```
extern int value3;
```

Notes

Two values associated with a variable
- The value stored at some location
  - rvalue
- The address in memory in which its data is stored
  - lvalue
- `theValue = value+1`
  - On right – data value
    - Data is read
  - On left - location value
    - Data is stored
- `theValue` is referred to as an object

Definition of a variable
- Causes storage to be allocated
- Introduces variable's name and type
- Optional initial value
  - `int number = 2;        // Declaration statement`

Declaration of a variable
- Announces variable exists and defined elsewhere
  - `extern int number;`
- Declaration is not a definition
  - Asserts definition exists elsewhere

Naming Conventions

Boolean	Type
TWindow, MZoom	Class
EDay	Enumeration type
fNumber	Data member
Draw	Member function
gApplication	Globals and static variable
TNote::fgUsers	Static data member
anArea	Automatic local variable Function arguments
kWindowId	Constant
aDrawArea	MultiWordNames

Notes

Type names must begin with a capital letter:

Class names begin with a T for base classes, and M for mixin classes

Enumeration type names should begin with an E.

Examples: Boolean, TView, MPrintable, EFreezeLevel. Avoid using C types directly

Members:

Data member names should begin with an f, for "field."

Member function names need only begin with a capital letter.

Example: fChanged, Draw().

Other:

Names of global variables (excluding static data members of classes) and static variables in functions should begin with a g

Example: gApplication.

Names of static data members (class globals) should begin with fg

Example: TView::fgClock.

Names of local variables (automatic only: statics are treated like globals, see above) and function arguments should begin with a word whose initial letter is lower case

Examples: seed, port, theArea.

Names of constants should begin with a k, including names of enumeration constants

Example: kMenuCommand.

In any name which contains more than one word, the first word should follow the convention for the type of the name, with the first letter of each word capitalized. Do not use underscores in names.

Examples: TContainerView (class name), fViewList (data member of class), fViewList (data member of class), RefreshSelf (function member of class), gDeviceList (global variable or local static), fgNumber (static data member), theArea (local or parameter)

Labs

- *Read all of the instructions before starting*
- We will be using MPW tools
- Compiling the exercises
    - Set the correct directory
    - ⌘ B or select Build from the Build menu
        - Type in ProgramName
    - Buildprogram ProgramName on the worksheet
- Run the program
    - ProgramName ➡Enter
- *Compare your solution to the solution*

Notes

Lab solutions are in a Solutions folder in each lab folder

Labs are designed to teach syntax

They are not application examples – certain things are not completely implemented

For example: error checking, memory management, …

When writing actual C++ applications you must pay attention to the same things you had to pay attention to when writing applications in other languages

In some cases you must pay even more attention to those things.

Section 2

User-Defined Types

Notes

This Section's Goals

- Demonstrate competence in:
  - Member operator functions
  - Non-member operator functions
  - User-defined conversions
  - Eliminating operator function and user-defined conversion ambiguity

Notes

Operator Functions

- C++ allows built in operators to be overloaded for user-defined types
- Operator functions

  ostream&   operator<<(const char*);
- cout << "hello world \n";

Notes

```
ostream&   operator<<(const char*);
ostream&   operator<<(int a);
ostream&   operator<<(long);
```

Operator overloading

The standard C operators can be overloaded for user-defined types

If you were to define a fixed point data type, you could define standard arithmetic operators for it

Use only where appropriate and clear

Defining the + operator for fixed point numbers helps clarify code

Defining & to mean "send a message" is crazy

Operator overloading only helps when the new operator is similar to the standard meaning of the built-in operator

What can be overloaded

Only predefined operators may be overloaded

Precedence or associativity cannot be changed

The unary/binary aspect cannot be changed

The overloaded instance must have at least one argument of the class type

This means that operators may only be defined for class types

There is only one instance of the ++ and -- operators (CFront 2.0)

Overloading does not distinguish between prefix and postfix

Defining both is likely to be ambiguous

The *signatures* must be distinct

Members or Non-Members

```
cout << stringStack  << "\n";
  Operand1     Operand2
ostream& operator<<(ostream& os, const String& str);
                    Non-member takes two arguments

stringStack[index];
                  Operand2
char& String::operator[](int index) {
    this -> ...          Member takes one less argument
Operand1 is implicit – the class object
```

Notes

Defining an operator overload as both member and non-member is ambiguous

Member or Non-Member

- Member operators are invoked only when an object of its class is the left operand

  stringStack[index];

  char& String::operator[](int index) {
- Non-members invoked based on signature

  cout << stringStack << "\n";

  ostream& operator<<(ostream& os, const String& str);

  It may have to be declared as a friend
- Required as class member functions:

  "=" "()" "->" "[]"

Notes

Non-member operator overloading is needed when implementing binary operators which can't be member functions

Only the left side of the expression is considered in operator overloading

A good example is the output stream `operator<<`

```
operator<< is the (overloaded) output operator for each built-in type
    ostream& operator<< (const char*);
    ostream& operator<< (int a);
    ...
```

The appropriate version of `operator<<` is called for each variable

```
ostream& operator<< (ostream& os, MyType& aMyType) {

    return (os<< AcceptableConversionOfMyType);
}
```

Required as class member functions:

Assignment operator "="

Function call operator "()"

Pointer member selector operator "->"

Array index operator "[]"

How to do cout << aString;

```
class String {
...
private:
    char*     fString;
    int       fLength;
};

ostream& operator<<(ostream& os, const String& str) {
    return (os << str.fString);
}
```

This shouldn't be allowed... should it?

Notes

Making friends

```
class String {
friend ostream& operator<<(ostream& os, const String& str);
...
private:
    char*     fString;
    int       fLength;
};

ostream& operator<<(ostream& os, const String& str) {
    return (os << str.fString);
}
```

Notes

`friend` classes and functions are in conflict with the ideas of encapsulation and independence
Avoid them except when implementing binary operators which can't be member functions
    Only the left side of the expression is considered in operator overloading
    Overloading `cout << MyClass`
        This cannot be a member
Accessibility
    Base class (inherited) member functions have no access to derived class members (unless declared a friend)
    Friends have no access to derived class members unless declared a friend to that derived class
    In general, friends have the same access privileges as the members of that class
Derivation vs. friendship
    Derivation extends the type
        Adding it own unique elements
    Friendship provides for access of non-public members
        There is no type relationship
    Derivation is not a special form of friendship
The `friend` declaration can be placed anywhere in the class definition
A `friend` is not able to use `this`.

Overloading operator[]

```
class String {
public:
    char&     operator[](int index);
    ...
};
```

Notes

Overloading operator[] in TString

```
void String::CheckIndex(int index) {... }

inline char& String::operator[](int index) {
    this->CheckIndex(index);
    return fString[index];
}

void main() {
    String* eString = new String("eString");
    int indx =0, aLen = eString->ReturnLength();
    for (indx; indx <aLen; ++indx)
        cout << (*eString)[indx];      // Or eString->operator[](index);
    cout << "\n";  └── First operand
}                      must be an object

eString
```

Notes

A Simple Point Class

```
class Point {
public:
                Point(short iV, short iH);
    void        Print();
private:
    short       v;
    short       h;
};
```

Notes

Lab 21

- In this lab you will overload the Stream class's operator<< to print a Point object

Notes

Lab 21 Output

aPoint (4,5) bPoint (2,3)
aPoint (4,5) bPoint (2,3)

Lab Directions

1. Set the directory to Lab 21.
2. Open Point.cp and Point.h.
3. Define an overloaded `operator<<` to print a `Point` object.
4. Make it a `friend` to the `Point` class.
5. Create two Points (`aPoint`) and (`bPoint`).
6. Include the following statement in `main()`:
```
cout << "aPoint " << aPoint  <<  " bPoint " << bPoint << "\n";
```
7. Compile and test the program.

Adding a New Type

I want a new type called `EInt`
This type stores itself encoded
It can be used anywhere an `int` is used

```
EInt aInt = 8;
```

Notes

class EInt

```
class EInt {
public:
                EInt(int theInt);
                EInt();
    void        PrintEInt();        For debugging
    void        PrintInt();         For debugging
private:
    void        EncodeInt(int theInt);
    int         DecodeInt();
    int         fInt;
};
```

Notes

Functions

```
EInt::EInt(int theInt) {
    this->EncodeInt(theInt);}
EInt::EInt() {
    this->EncodeInt(0);}
void EInt::EncodeInt(int theInt) {
   fInt = theInt+4;}
int EInt::DecodeInt() const {
    return fInt-4;}
void EInt::PrintEInt() const {
    cout << "The encoded int: " << fInt << "\n";}
void EInt::PrintInt() const {
    cout << "The decoded int: " << this->DecodeInt() << "\n";}
```

Notes

Results

```
void
main() {

    EInt eInt = 8;

    eInt.PrintEInt();
    eInt.PrintInt();
}
```

The encoded int is 12
The decoded int is 8

Notes

Some Problems With EInt

```
void
main() {
    EInt int1;
    EInt int2;
    EInt int3;
```

To add two EInts

```
    int1 = AddEInt(int2,int3);
```

If EInt were really as easy to use as a built-in type, we should be able to:

```
    int1 = int2+int3;
}
```

Notes

But our encoded integer is not very easy to use

Typical operations on encoded integers must be coded as functions

This is awkward

Point Example

```
void
main() {

Point aPoint(4,4);
Point bPoint(2,2);
Point cPoint;
Point dPoint;

cPoint = aPoint + bPoint;
dPoint = aPoint – bPoint;

cout << "cPoint " << cPoint << "\n";
cout << "dPoint " << dPoint << "\n";
}
```

Notes

Member Overload operator+

```
Point Point::operator+(const Point& pt) const {
    return Point(v + pt.v, h + pt.h); // Only one Point created
}
```

Notes

Lab 22

- In this lab you will overload the basic arithmetic operators +, -, /, * as members of the EInt class

Notes

Lab 22 Output

aInt1
The encoded int is 12
The decoded int is 8
aInt2
The encoded int is 13
The decoded int is 9
aInt2+aInt1
aInt3
The encoded int is 21
The decoded int is 17
aInt2-aInt1
aInt3
The encoded int is 5
The decoded int is 1
aInt2/aInt1
aInt3
The encoded int is 5
The decoded int is 1
aInt2*aInt1
aInt3
The encoded int is 76
The decoded int is 72

Lab Directions

1. Set the directory to Lab 22.
2. Open EInt.cp and EInt.h.
3. Overload the `+`, `-`, `*`, and `/` operators for `EInt`, as members.
4. Define 3 `EInt`'s
   - `aInt1` initialized to 8,
   - `aInt2` initialized to 9,
   - `aInt3` initialized to 0.
5. Print the encoded and decoded values of `aInt1` and `aInt2`.
6. Add `aInt1` to `aInt2`, and assign the result to `aInt3`.
7. Print the encoded and decoded value of the `aInt3`.
8. Subtract `aInt1` from `aInt2`, and assign the result to `aInt3`.
9. Print the encoded and decoded value of `aInt3`.
10. Divide `aInt2` by `aInt1` and assign the result to `aInt3`.
11. Print the encoded and decoded value of `aInt3`.
12. Multiply `aInt1` by `aInt2`, and assign the result to `aInt3`.
13. Print the encoded and decoded value of `aInt3`.
14. Compile and test the program.

What Doesn't Work

```
void main() {

    EInt            aInt1 = 0;
    EInt            aInt2 = 9;

    aInt1 = 2 + aInt2;

}

# error: bad operand types int  EInt  for +
```

Notes

Non-Member Overload operator-

```
Point operator–(const Point& pt1, const Point& pt2) {
    return Point(pt1.v - pt2.v, pt1.h - pt2.h)
}
```

Notes

Lab 23

- In this lab you will overload the basic arithmetic operators +, -, /, * as non-members of the EInt class

Notes

Lab 23 Output

aInt1
The encoded int is 12
The decoded int is 8
aInt2
The encoded int is 13
The decoded int is 9
aInt2+aInt1
aInt3
The encoded int is 21
The decoded int is 17
aInt2-aInt1
aInt3
The encoded int is 5
The decoded int is 1
aInt2/aInt1
aInt3
The encoded int is 5
The decoded int is 1
aInt2*aInt1
aInt3
The encoded int is 76
The decoded int is 72

Lab Directions

1. Set the directory to Lab 23.
2. Open EInt.cp and Eint.h.
3. Overload the `+`, `-`, `*`, and `/` operators for `EInt` as non-members.
4. Make the operators `friend` functions of `EInt`.
5. Define 3 EInt's
   `aInt1` initialized to 8,
   `aInt2` initialized to 9,
   `aInt3` initialized to 0.
6. Print the encoded and decoded values of `aInt1` and `aInt2`.
7. Add `aInt1` to `aInt2`, and assign the result to `aInt3`.
8. Print the encoded and decoded value of the `aInt3`.
9. Subtract `aInt1` from `aInt2`, and assign the result to `aInt3`.
10. Print the encoded and decoded value of `aInt3`.
11. Divide `aInt2` by `aInt1` and assign the result to `aInt3`.
12. Print the encoded and decoded value of `aInt3`.
13. Multiply `aInt1` by `aInt2`, and assign the result to `aInt3`.
14. Print the encoded and decoded value of `aInt3`.
15. Compile and test the program.

Some More Considerations

```
void main() {

    EInt            aInt1 = 8;
    EInt            aInt2 = 9;
    EInt            aInt3 = 0;

    aInt1 = aInt2 + 2;  ─┐
    aInt3 = 2 + aInt2;  ─┴ Now work fine ... but ...

    int aInt4 = aInt3;
}
```

error: bad initializer type EInt for aInt4 (int expected)

Notes

User-Defined Conversions

```
void printString(const String& aStr) { ... }

class String {
public:
    String(char* string);          Converts char* to String
    operator char* ();             Converts String to char*
...};

void main() {
    String     stringObject = String("stringObject");
    char* aCharPtr = "Hello";
    aCharPtr = stringObject;
    printString(aCharPtr);
}
```

Notes

- Type conversion
  - Standard conversions limits the number of operators and overloaded conversions for built-in types
    - `char`, `short`, and `int` can all be automatically converted in expressions
      - It is unnecessary to define
      - f(`int`);
      - f(`char`);
      - f(`short`)
    - They are all promoted to `int`
    - Only operations on `int` then need be defined
    - Type conversion is done by the compiler and is transparent to the user
- User-defined type conversions
  - Allow us to define set of conversions that can be applied to members of that class
  - Inform the compiler how that conversion is to be done
  - How to convert from *this user-defined type to another type*
- Single argument constructors
  - How to convert from *a type to this user-defined type*

Conversion Functions

```
String::String(char* string) {      Single argument constructors
    fLength = strlen(string);
    fString = new char[fLength + 1];
    strcpy(fString, string);
    fLastIndex =0;
}

String::operator char* () {          User-defined type conversions
    return fString;
}
```

Notes

Conversions

- Standard conversions will be done before user-defined ones
- User-defined conversion operators are utilized last
- User-defined conversion operators are allowed for built-in, derived, or class types
- Only one level of user-defined conversions can apply
  - Standard -> User-defined -> Standard ... is allowed

Notes

Standard conversions will be done before user-defined ones
- User-defined conversions are called only if no other conversions are possible

Conversion operators are called only if there is no other way to do it
- Other overloaded functions
- Assignment operators

Conversion operators are allowed for built-in, derived, or class types
- Not for arrays or functions
- Must be a member function
- Multiple user-defined conversions achieving a match is ambiguous
  - Conversion constructors and operator conversions share the same precedence

Conversion operators are inherited

What if the required type does not exactly match any of the conversion operator types?
- Standard conversion used to find user-defined conversion
- Standard conversions applied to user-defined conversions
- Will not allow a second user-defined conversion
  - Only one level of user-defined conversions can apply

Overloaded functions with class arguments
- There is no distinction between a class object and a reference
- Standard conversion
  - Derived class object, reference, or pointer implicitly converted into public base class type
  - A pointer to any class type converted to void*
  - Typed to the type of the function with the "closest" base class

Ambiguity of Conversions

```
class String {
public:
    operator short ();
    operator long ();
... };

void printDay(int aDay){ ...}

void main() {
    String    stringObject1 ="1";
    printDay(stringObject1);
}

error:  2 possible conversions for argument
```

Notes

Ambiguity can result from application of conversions

Often an explicit cast will resolve the ambiguity

If two conversion operators are possible, and one is an exact match, while the other requires a standard conversion, there is no ambiguity

Explicit Cast

```
void main() {
    String    stringObject1 ="1";
    printDay(short(stringObject1));
}
```

Notes

Lab 24

- In this lab you will define conversion operators that allow an EInt to be converted to int, char, or short

Notes

Lab 24 Output

EInt::operator Type() EInt::operator Type() EInt::operator Type() AllowEntry Entry denied	Output of step 8
EInt::operator char() EInt::operator short() EInt::operator int() AllowEntry Entry denied	Output of step 11

Lab Directions

1. Set the directory to Lab 24.
2. Open EInt.cp and EInt.h.
3. Examine the operator overloads.
4. Examine Security`::AllowEntry(...)`. Notice it takes three arguments – `int`, `short`, and `char`.
5. Write a single conversion that allows us to pass in an `EInt` to `Security::AllowEntry(...)`. Place a `cout` statement in the conversion function to know it has been called.
6. Define a `Security` object, `aSecurity` on the stack.
7. Define three `EInt`'s, `aInt1` initialized with 8, `aInt2` initialized with 9, and `aInt3` initialized with 0. Call `Security::AllowEntry(...)` with `aInt1`, `aInt2`, and `aInt3` as arguments.
8. Compile and test the program.
9. Define two more conversions so that all three conversions (`int`, `char`, and `short`) are defined. Place a `cout` statement in each of the conversion functions to know which has been called.
10. Call `Security::AllowEntry(...)` with three `EInt`'s as arguments.
11. Compile and test the program.

Lab 25

- In this lab you will define a set of operator overloads and user-defined conversions that eliminate ambiguity

Notes

Lab 25 Output

realInt = 1
realShort = 2

aInt3 = aInt2 + realInt
aInt3: Encoded = 14 Decoded = 10

aInt3 = aInt2 + realShort
aInt3: Encoded = 15 Decoded = 11

realInt = aInt2 + realShort
EInt::operator int()
realInt = 11

realShort = realInt + aInt2
EInt::operator short()
realShort = 20

Lab Directions

1. Set the directory to Lab 25.
2. Open EInt.cp and EInt.h.
3. Examine the operator overloads.
4. Compile the program.
5. Modify your operator overloads to generate the required output.
6. For hints, see the Hint file.
7. Compile and test the program.

Section 3

Initialization and Assignment

Notes

This Section's Goals

- Demonstrate competence in:
  - Knowing when and how to use default initialization
  - Defining an initialization constructor
  - Using the member initialization list to initialize a base class and member class object
  - Knowing when and how to use default assignment
  - Overloading an assignment operator
  - Invoking base class and member class object assignment operators in an overloaded assignment operator

Notes

Initialization and Assignment

- A distinction that may be unimportant
- Initialization

    An instance of a class is created

    Done in a constructor of the type:

    ```
    X::X(const X&)
    EInt aInt1 = aInt2;
    ```

- Assignment

    An object is replaced by another object

    Done in an assignment operator of the type:

    ```
    X& X::operator= (const X&)
    aInt1 = aInt2;
    ```

Notes

Lab 26

- In this lab you will initialize one object with another, and examine the default compiler behavior

Notes

Lab 26 Output

aLab1Comment: Great lab
aLab2Comment: Great lab

Lab Directions

1. Set the directory to Lab 26.
2. Open Comment.cp and Comment.h.
3. Notice the `Comment` class has one constructor, and it takes a string as an argument.
4. Create one `Comment` object (`aLab1Comment`) on the stack, initializing it with a string.
5. Create a second `Comment` object (`aLab2Comment`) on the stack, initializing it with `aLab1Comment`. What do you thing will happen?
6. Print both `Comment` objects to check your results.
7. Compile and test the program.

Initialization

```
EInt        aInt1 = 8;
EInt        aInt2 = aInt1;

aInt1.PrintEInt();
The encoded int is 12
aInt1.PrintInt();
The decoded int is 8

aInt2.PrintEInt();
The encoded int is 12
aInt2.PrintInt();
The decoded int is 8
```

Notes

Memberwise Initialization

- The "usual" constructors are not invoked when initializing one class object with another
- The initialization of `aInt2` is done through copying each element of `aInt1` into `aInt2`

  This is called memberwise initialization
- The compiler generates a constructor of the type X::X(const X&);

```
EInt::EInt(const EInt& aEInt){
    fInt = aEInt.fInt;
}
```

Notes

An Easy "Mistake"

```
TString::TString(char* theString) {
    cout << this << " In constructor with string \n"; ...

TString::TString(TString& theString){
    cout << this << " In initialization constructor with string \n";...

char* operator+ (TString string1, TString string2) {
cout <<  " In plus operator\n"; ...

void main() {
    ...
    TString            fString(aString+bString);
}
```

In initialization constructor with string
In initialization constructor with string
In plus operator
In constructor with string

Notes

Memberwise Initialization Happens:

```
TString     bString("Hello world");
TString     aString(bString);

char*
Compare (TString string1, TString string2)

TString
TString::operator+ (TString& string2)
```

Member class objects are not copied
Memberwise initialization is recursively applied

Notes

Memberwise initialization
- Copies each built-in or derived from built-in type data member
- Member objects are not copied
  - Memberwise initialization is recursively applied

Memberwise initialization occurs when:

1. One class object is initialized with another
```
TString bString("Hello world");
TString aString(bString);
```
2. A class object is passed as an argument to a function
```
char* operator- (TString string1, TString string2);
```
3. A class object is the return value of a function
```
TString  TString::operator+ (TString& string2);
```


No Memberwise Initialization

char*
TString::operator+ (TString& string2)

TString&
TString::operator+ (TString& string2)

Notes

There is no memberwise initialization when:

1. A class object is passed as a reference argument to a function
   ```
   char* TString::operator+ (TString& string2);
   ```
2. A class object reference is the return value of a function
   ```
   TString& TString::operator+ (TString& string2);
   ```

Explicit Initialization Constructors

```
class EInt {
...
    int *⇦          fInt;     };

void main() {
    EInt            aInt1 = 8;
    EInt            aInt2 = aInt1;

    aInt1.PrintEInt();
    aInt2.PrintEInt();
}
```

The encoded int is 12 The fInt = 0x157e44
The encoded int is 12 The fInt = 0x157e44

Notes

Consequences of memberwise initialization
- At destructor time, the same pointer will be deleted twice
  - aInt1
    - `fInt =0x157e44`
  - aInt2
    - `fInt = 0x157e44`
- There will be a problem if you try to delete the pointer twice

Solution: the X(const X&) constructor
- An explicit initialization constructor
- When defined it is invoked for each initialization of one class object with another.
- `EInt::(EInt& theEInt)` is invoked and each allocates a new `fInt` so that each fInt has its own area of memory.

Point Constructors

```
inline Point::Point(short iV, short iH) {
    v = iV;
    h = iH;
}

inline Point::Point(const Point& pt) {
    v = pt.v;
    h = pt.h;
}
```

Copy constructors often do the same thing "regular" constructors do

Notes

Lab 27

- In this lab you will determine when default memberwise initialization should not be done, and define the necessary initialization constructor

Notes

Lab 27 Output

aLab1Comment: Great lab
aLab2Comment: Great lab
fText deleted
fText deleted

Lab Directions

1. Set the directory to Lab 27.
2. Open Comment.cp and Comment.h.
3. Make `Comment::fText` a `char*`.
4. In the constructor allocate memory using `new`, and copy the string argument into that memory.
   Is this something you would normally want to do?
5. Define a `Comment::~Comment()` destructor.
   In it delete fText.
   Put a cout statement in the destructor to let you know that it does execute.
6. Create one `Comment` object (`aLab1Comment`) on the stack, initializing it with a string.
7. Create a second `Comment` object (`aLab2Comment`) on the stack, initializing it with `aLab1Comment`.
8. Compile the program.
   What do you think will happen when the program finishes executing?
9. Before you execute the program, ***Save Your Work***
10. Run the program, **g stoptool** or **g sysrecover** will often help.
11. Modify your program so that it executes correctly.

Members and Base Classes

- EPoint
  - Like the Point class except:
    - Point::v and Point::h are now of type EInt (instead of short)
- EInt
  - fInt is of type int*
- Point
  - Derived from EPoint
    - Additional data members fId of type EInt and fName of type char*

Notes

The Classes

EInt Copy constructor needed

int* fint;

EPoint Copy constructor not needed

EInt v;

EInt h;

Point Copy constructor needed

EInt fId;

char* fName;

Notes

EPoint Has an EInt Member Class Object

```
class EPoint {
public:
    EPoint(short iV, short iH);
    // Needs no initialization constructor
    ...
private:
    EInt v;
    EInt h;
};
```

Notes

EInt – Point's Member Class Object

```
class EInt {
public:
    EInt(int theInt);
    EInt(const EInt& aInt);    Needs initialization constructor
    ~EInt(); ——delete fInt;
    ...
private:
    ...
    int*      fInt;
};
```

Notes

A Class Hierarchy

```
class EPoint {
public:
     EPoint(short iV, short iH);
     ...
private:
     EInt v;
     EInt h;  };

class Point : public EPoint {
public:
     Point(short iV,  short iH, char* aId);
     Point(const Point& pt);          Needs initialization constructor
     ~Point();————delete fName;————————┘
     ...                       |
private:           ┌───────────┘
     char*        fName;
     EInt         fId };
```

Notes

The Program

```
void main() {
    Point aPoint(4, 4, 1, "aPoint");
    Point bPoint(2, 2, 10, "bPoint");
    cout << "aPoint " << aPoint << "\n";
    cout << "bPoint " << bPoint << "\n";
    Point cPoint(aPoint);
    cout << "cPoint " << cPoint << "\n";
...}
```

aPoint (4,4) Id = 1 fName = aPoint

bPoint (2,2) Id = 10 fName = bPoint

cPoint (4,4) Id = 1 fName = aPoint+1

Notes

Initialization Requirements?

- Point needs an initialization constructor because of fName
- EInt needs an initialization constructor because of fInt
- EPoint appears not to need an initialization constructor

Got it?

Notes

Default Initialization Process

- Base classes are recursively memberwise initialized before derived classes
  - In order of base class declaration
- Member class objects are recursively memberwise initialized before containing classes
  - In order of member class declarations

EPoint::EInt

EPoint

Point::EInt

Point

Notes

Member Class Object Initialization

- If there is a Point::Point(const Point&)
  - Invoke member initialization list
  - For class member objects not in the member initialization list that require a constructor
    - Invoke constructor that takes no arguments
    - If there is none – error
  - Invoke Point(const Point&)
- If there is no Point::Point(const Point&)
  - Perform recursive memberwise initialization for member class objects

Notes

Initialization Responsibilities

- The containing class does not define a X(const X&) constructor and a member class object does
  - That member class object's X(const X&) is invoked
  - Other member class objects may be memberwise initialized
  - The containing class object is memberwise initialized
- The containing class does define a X(const X&) constructor
  - Member class object initialization is the responsibility of the containing class's initialization constructor
    - Or a constructor with no arguments is invoked
- Point must invoke EInt's initialization constructor!

Notes

Handling of the member class initialization becomes responsibility of the containing class constructor

```
ContainingClass(const ContainingClass& aContainingClass ): MemberClass(...)
```

If there is no ContainingClass(ContainingClass&)

Memberwise initialization is done

Point's Constructor

```
class Point : public EPoint {
    ...
    Point(const Point& pt);  — Needed because of fName ...
                               also becomes responsible for fId
private:
    EInt     fId;
    char*    fName;
};
```

Notes

Initializing fId

Member initialization list

```
Point::Point( const Point&  pt) : fId(pt.fId)
{ ... }

class Point {
public:
    Point( const Point& pt);
    ...
private:
    EInt      fId;  EInt(const EInt& aInt) invoked for fId
    ...
};
```

Notes

Member initialization list

- Follows constructor signature and set off with a colon followed by a comma-separated list of member name/argument pairs
- Each member may appear once
- Can appear only in the definition of the constructor
- Data members that are built-in types may also be initialized

Base Class Initialization

- If there is a Point::Point(const Point&)
  - Invoke member initialization list
  - For base classes not in the member initialization list that require a constructor
    - Invoke the constructor that takes no arguments
    - If there is none – error
  - Invoke Point(const Point&)
- If there is no Point::Point(const Point&)
  - Perform recursive base class memberwise initialization

Notes

Initialization Responsibilities

- The derived class does not define a X(const X&) constructor and a base class does
  - The base class's X(const X&) is invoked
  - The derived class object is memberwise initialized
- The derived class does define a X(const X&) constructor
  - Base class initialization is the responsibility of the derived class's initialization constructor
    - Or a constructor with no arguments is invoked
- Point is responsible for initializing EPoint!

Notes

Handling of the base class initialization becomes responsibility of the derived class constructor

```
DerivedClass(const DerivedClass& aDerivedClass ): BaseClass(aDerivedClass)
```

If there is no BaseClass(BaseClass&)

Memberwise initialization is done

Point's Constructor

```
class Point : public EPoint {
    ...                                      Needed because of fName ...
    Point(const Point& pt);  –  became responsible for fId ...
private:                                     also responsible for EPoint
    EInt      fId;
    char*     fName;
};
```

Notes

Initializing EPoint

Member initialization list

```
Point::Point(const Point& pt) : EPoint(pt), fld(pt.fld)
{...}

class EPoint {          No initialization constructors ...
    ...                 memberwise initialization
private:
    EInt v;
    EInt h;             EInt(const EInt& aInt); invoked
};
```

Notes

Default Initialization

No base class initialization specified in member initialization list ...

```
Point::Point(const Point& pt)  :/* EPoint(pt), */ fld(pt.fld)
{...}

class EPoint {
public:
    EPoint();        EPoint::EPoint() : v(0), h(0)  invoked
    ...
private:
    EInt v;
    EInt h;          EInt::EInt(0)  invoked
};
```

Notes

The Message

- The distinction between initialization and assignment phases again becomes important
  The other time was for const and references
- For base and member class initialization constructors to be invoked:
  They must be in the member initialization list

Notes

Lab 28

- In this lab you will define an initialization constructor that uses the member initialization list to initialize its base class and member class object

Notes

Lab 28 Output

```
aLab1CategorizedComment:
  The category: Morning
  The comment: Great lab
  The owner: Joe
aLab2CategorizedComment:
  The category: Morning
  The comment: Great lab
  The owner: Joe
aLab3CategorizedComment:
  The category: Afternoon
  The comment: I'm learning a lot
  The owner: John
fOwner deleted
fText deleted
fText deleted
fOwner deleted
fText deleted
fText deleted
fOwner deleted
fText deleted
fText deleted
```

Lab Directions

1. Set the directory to Lab 28.
2. Open CategorizedComment.cp and CategorizedComment.h.
3. Notice `CategorizedComment` is a class `private`ly derived from `Comment` with a `Comment` as a member.
   It is `private` because it is not a "kind of" `Comment`.
4. Explain why it requires an explicit initialization constructor
5. Define an initialization constructor; what is it responsible for initializing?
6. Create one `CategorizedComment` object (`aLab1CategorizedComment`) on the stack, initializing it with a string.
7. Create a second `CategorizedComment` object (`aLab2CategorizedComment`) on the stack, initializing it with `aLab1CategorizedComment`.
8. Create a third `CategorizedComment` object (`aLab3CategorizedComment`) on the stack, initializing it with a string.
9. Print `aLab1CategorizedComment`, `aLab2CategorizedComment`, and `aLab3CategorizedComment`.
10. How many times should the `Comment` destructor be called?
    Place a cout statement in `Comment::~Comment()` to check your answer.
11. Compile and test the program.

Lab 29

- In this lab you will assign one object to another, and examine the default compiler behavior

Notes

Lab 29 Output

aLab1Comment: Great lab
aLab2Comment: Another Lab
aLab2Comment = aLab1Comment
aLab1Comment: Great lab
aLab2Comment: Great lab

Lab Directions

1. Set the directory to Lab 29.
2. Open Comment.cp and Comment.h.
3. Notice the `Comment` class has one constructor, and it takes a string as an argument.
4. Create one `Comment` object (`aLab1Comment`) on the stack, initializing it with a string.
5. Create one `Comment` object (`aLab2Comment`) on the stack, initializing it with a string.
6. Print both `Comment` objects.
7. Assign `aLab1Comment` to `aLab2Comment`.
   What do you think will happen?
8. Print both `Comment` objects to check your answer.
9. Compile and test the program.

Assignment

```
EInt        aInt1 = 8;
EInt        aInt2 = 7;

aInt1.PrintEInt();
The encoded int is 12

aInt2.PrintEInt();
The encoded int is 11

aInt1 = aInt2;

aInt1.PrintEInt();
The encoded int is 11
```

Notes

Memberwise Assignment

- The mechanics of assignment of one class object to another of its class is the same as memberwise initialization
- Memberwise assignment of non-static data members
- The compiler generates a special assignment operator of the type

  X& X::operator= (const X&);

  It is not inherited

Notes

Explicit Assignment Operators

```
class EInt { ...
    int *⇦        fInt;     };

    EInt          aInt1 = 8;
    EInt          aInt2 = 7;

    aInt1.PrintEInt();
    aInt2.PrintEInt();
    aInt1 = aInt2;
    aInt1.PrintEInt();
```

The encoded int is 12 The fInt = 0x157e6c
The encoded int is 11 The fInt = 0x157e78
The encoded int is 11 The fInt = 0x157e78

Notes

Consequences

At destructor time, the same pointer will be deleted twice

aInt1

`fInt = 0x157e78`

aInt2

`fInt =  0x157e78`

There will be a problem if you try to delete the pointer twice.

The storage previously allocated to fInt in aInt1 is lost forever;

`fInt = 0x157e6c`

There could have been other state variables we would have wanted initialized.

Order of creation.

Indexes (as in TString).

Solution: X& X::operator= (const X&);

We can provide (as with initialization) an explicit

`X& X::operator= (const X&);`

An explicit assignment operator.

When defined it is invoked for each assignment of one class object to another

It simply copies the new value into *fInt

String Assignment

```
String&
String::operator = (const String& aString) {
    if (this == &aString) return *this;
    delete fString;
    fLength = aString.fLength;
    fString = new char[fLength+1];
    strcpy(fString, aString.fString);
    return *this;
}
```

Assignment operators often do what initialization constructors do ... plus a little more

Notes

Lab 30

- In this lab you will determine when default memberwise assignment should not be done, and define the necessary assignment operator overload

Notes

Lab 30 Output

aLab1Comment: Great lab
aLab2Comment: Another lab
aLab2Comment = aLab1Comment
aLab1Comment: Great lab
aLab2Comment: Great lab
fText deleted
fText deleted

Lab Directions

1. Set the directory to Lab 30.
2. Open Comment.cp and Comment.h.
3. Make `Comment::fText` a `char*`.
4. In the constructor allocate memory on the heap for it, and copy the string argument into that memory.
5. Define a `Comment::~Comment()` destructor that `deletes` the `fText` member.
   Put a `cout` statement in the destructor to let you know that it does execute.
6. Create one `Comment` object (`aLab1Comment`) on the stack, initializing it with a string.
7. Create a second `Comment` object (`aLab2Comment`) on the stack, initializing it with a string.
8. Assign `aLab1Comment` to `aLab2Comment`.
9. Compile the program.
   What do you think will happen when the program finishes executing?
10. Before you execute the program, ***Save Your Work.***
11. Run the program, **g stoptool** or **g sysrecover** will often help.
12. Modify your program so that it executes correctly.

Members and Base Classes

- EPoint
  - Like the Point class except:
    - Point::v and Point::h are now of type EInt (instead of short)
- EInt
  - fInt is of type int*
- Point
  - Derived from EPoint
    - Additional data members fId of type EInt and fName of type char*

Notes

The Classes

EInt Assignment operator needed

```
int*    fInt;
```

EPoint Assignment operator not needed

```
EInt    v;
EInt    h;
```

Point Assignment operator needed

```
EInt    fId;
char*   fName;
```

Notes

EPoint Has an EInt Member Class Object

```
class EPoint {
public:
    EPoint(short iV, short iH);
    // Needs no assignment operator
...
private:
    EInt v;
    EInt h;
};
```

Notes

EInt – Point's Member Class Object

```
class EInt {
public:
    EInt(int theInt);
    EInt& operator=(const EInt& aInt);
    EInt& operator=(int aInt);
    ~EInt(); ——delete fInt; ———— Needs assignment operator
    ...
private:
    ...
    int*        fInt;
};
```

Notes

A Class Hierarchy

```
class EPoint {
public:
    EPoint(short iV, short iH);
...
private:
    EInt v;
    EInt h;  };

class Point : private EPoint {
public:
    Point(short iV,  short iH, char* aId);
    Point& operator=(const Point& pt);      Needs assignment operator
    ~Point();————delete fName;——————————┘
...
private:
    char*      fName;
    EInt       fId };
```

Notes

The Program

```
void main() {
    Point aPoint(4, 4, 1, "aPoint");
    Point bPoint(2, 2, 10, "bPoint");
    cout << "aPoint " << aPoint << "\n";
    cout << "bPoint " << bPoint << "\n";
    aPoint = bPoint;
    cout << "aPoint " << aPoint << "\n";
...}
```

aPoint (4,4) Id = 1 fName = aPoint

bPoint (2,2) Id = 10 fName = bPoint

aPoint (2,2) Id = 10 fName = bPoint+1

Notes

Assignment Requirements?

- Point needs an assignment operator because of fName
- EInt needs an assignment operator because of fInt
- EPoint appears not to need an assignment operator

Got it?

Notes

Default Assignment Process

- Class type of assignment determined by the class of the object on the left side of the =
- Member class objects are recursively memberwise assigned
- Base classes are recursively memberwise assigned

Notes

Member Class Object Assignment

- If there is a Point::operator=(const Point& pt)

  Do nothing
- If there is no Point::operator=(const Point& pt)

  Perform recursive member class object memberwise assignment

Notes

Assignment Responsibilities

- The containing class does not define an operator= and a member class object does
  - That member class object's operator= is invoked
  - Other member classes objects may be memberwise assigned
  - The containing class object is memberwise assigned
- The containing class does define an operator=
  - Member class assignment is the responsibility of the containing class's operator=
    - Or no assignment is done
- Point must invoke EInt's assignment operator!

Notes

Point's Assignment Operator

```
class Point : public EPoint {
public:
    Point& operator = (const Point& pt);
    ...
private:
    EInt     fId;
    char*    fName;
};
```

Needed because of fName ...
also becomes responsible for fId

Notes

Assigning fId

```
Point& Point::operator = ( const Point& pt) {
    ...
    fId = pt.fId;
...}


class Point : public EPoint {
public:
    Point& operator = (const Point& pt);
    ...
private:
     EInt      fId;    EInt& operator=(const EInt& aInt) invoked
     ...
};
```

Notes

Base Class Assignment

- If there is a Point::operator=(const Point& pt)
  Do nothing
- If there is no Point::operator=(const Point& pt)
  Perform recursive base class memberwise assignment

Notes

Assignment Responsibilities

- The derived class does not define an operator= and a base class does
  - The base class's operator= is invoked
  - The derived class is memberwise assigned
- The derived class does define an operator=
  - Base class assignment is the responsibility of the derived class's operator=
    - Or no assignment is done
- Point must invoke EPoint's assignment operator!

Notes

Point's Assignment Operator

```
class Point : public EPoint {
public:
    Point& operator = (const Point& pt);
    ...
private:
    EInt      fId;
    char*     fName;
};
```

Needed because of fName ...
became responsible for fId ...
also responsible for EPoint

Notes

Assigning EPoint

```
Point& Point::operator = (const Point& pt) {
    ...
    this->EPoint::operator=(pt);
...}

class EPoint {          No assignment operator ...
    ...                 memberwise assignment
private:
    EInt v;
    EInt h;        EInt& operator=(const EInt& aInt); invoked
};
```

Notes

Lab 31

- In this lab you will define an assignment operator that invokes its base class's and member class object's assignment operators

Notes

Lab 31 Output

```
aLab1CategorizedComment:                 fOwner deleted
  The category: Morning                  fText deleted
  The comment: Great lab                 fText deleted
  The owner: Joe                         fOwner deleted
aLab2CategorizedComment:                 fText deleted
  The category: Afternoon                fText deleted
  The comment: Another lab
  The owner: Joe
aLab2CategorizedComment = aLab1CategorizedComment
aLab1CategorizedComment:
  The category: Morning
  The comment: Great lab
  The owner: Joe
aLab2CategorizedComment:
  The category: Morning
  The comment: Great lab
  The owner: Joe
```

Lab Directions

1. Set the directory to Lab 31.
2. Open CategorizedComment.cp and CategorizedComment.h.
3. Notice `CategorizedComment` is a class privately derived from `Comment` with a `Comment` as a member. It is private because it is not a "kind of" `Comment`.
4. Explain why it requires an assignment operator.
5. Define an assignment operator; what is it responsible for?
6. Create one `CategorizedComment` object (`aLab1CategorizedComment`) on the stack, initializing it with a string.
7. Create a second `CategorizedComment` object (`aLab2CategorizedComment`) on the stack, initializing it with a string.
8. Print `aLab1CategorizedComment and aLab2CategorizedComment`.
9. Assign `aLab1CategorizedComment` to `aLab2CategorizedComment`.
10. Print `aLab1CategorizedComment and aLab2CategorizedComment`.
11. Compile and test the program.

Initialization and Assignment Guidelines

- If a class provides either an assignment operator or an initialization constructor, it will probably need both
- They are required when:
  - Objects allocate memory or create objects for their exclusive use
  - There are state dependent data members

Notes

Section 4

Inheritence

Notes

This Section's Goals

- Demonstrate competence in:
  - Knowing how and when to use public and private inheritance
  - Using multiple inheritance and resolving ambiguity
  - Using virtual base classes to eliminate ambiguity
  - Understanding what happens when casting a pointer to a class object with multiple bases

Notes

Public or Private Inheritance

- Base classes should be public when type information is required
  - Usually only useful for polymorphism
    - A function expecting a TStudent is handed a TGradStudent
- Base classes should be private unless there is a reason to share the base class's public protocol
  - The base class's behavior is inherited (and may be overridden)
  - Portions of a base class's protocol can be shared by reexporting the necessary members as public

Notes

Using Public Inheritance

- Normal inheritance
  - Strictly extending the type – Add functionality and:
    - Implement behavior
    - Associate values with abstract properties
    - Maintain a type relationship
- Non-normal inheritance
  - For simple code reuse or design problems
    - Complete overrides are possible
    - Change interface
    - Use of private inheritance

Notes

Normal inheritance
- Extending the type – Add functionality and:
  - Implement behavior
    - For unimplemented abstract functions
    - Incrementally improve implemented functions
      - Always calling the inherited implementation
  - Associate values with abstract properties
    - Size is an abstraction
    - 3-8 feet tall is an implementation
  - Maintain a type relationship
    - All members share the abstraction's properties
- Goal – maintain purity of abstractions

Inheritance Hierarchy

Class

↑ Non-normal

Class

aDerived Class

↑ Normal

Class

aDerived Class

bDerived Class

Notes

Assemble normal bases

- Inherit from a variety of classes, that match, or closely match, what we need
- If necessary, completely override some behavior, and add some
- Then treat this as a normal base

Using Private Inheritance

- The derived class is not considered a subtype

  The subclass is not committed to act as a subtype

- The derived class has access to all base class members

  The relationship between a privately derived class and its base class is the same as the relationship between a publically derived class and its base class

  ... but base class members are not publicly exported as members of the derived class

- Ther is no automatic derived class to base class conversion

  Function arguments or assignments

Notes

Inheritance Should Not Be Used

- If a class is used by another class in purely a client relationship

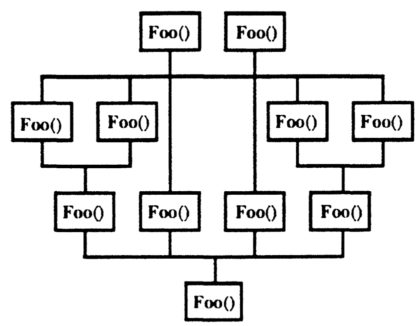
  A pointer to that class should be a member

Notes

Multiple Inheritance

class TGradStudent : public TStudent , public TFaculty {...

- Use in a controlled fashion
- Can result in a "write only" class structures



Notes

A Multiple Inheritance Strategy

- Define two categories of classes
  - Base classes which represent fundamental functional objects (like a car)
  - Mixin classes which represent optional functionality (like power steering)
- There are two rules you should follow

Notes

Rules for Multiple Inheritance

- Rule 1 – A class can inherit from zero or one base classes, plus zero or more mixin classes

  If a class does not inherit from a base class, it really may be a mixin class

  But not always: at the root of a hierarchy for example

- Rule 2 – A class which inherits from a base class is a base class

  It cannot be a mixin class

  Mixin classes can only inherit from other mixin classes

Notes

The Net Effect of the Two Rules

- Base classes form a conventional, tree-structured inheritance hierarchy rather than an arbitrary acyclic graph
  - The base class hierarchy becomes much easier to understand
  - Mixins then become add-in "options" which do not fundamentally alter the inheritance hierarchy
- But sometimes you should ignore the rules
  - Multiple inheritance can and should be used in other ways if it makes sense

Notes

(Multiple) Inheritance

- Things to consider

  Is there an "is-a" (subtype) relationship (for public inheritance)?

  Is it necessary to modify behavior (override functions)?

  Will polymorphism be used?

Notes

Multiple Inheritance Ambiguity

```
class TDirectoryEntry {
public:
    char*       ReturnName();
private:
    char        fName[20];
};

class TFaculty: public TDirectoryEntry {...};
class TStudent: public TDirectoryEntry {...};

class TGradStudent: public TStudent , public TFaculty { ...
    virtual void        PrintAdvisee(); ...};

void   TGradStudent::PrintAdvisee() {
    cout << "\n" << this->ReturnName() ... }

error: ambiguous TStudent::ReturnName() and TFaculty::ReturnName()
```

Notes

Shouldn't often be a problem if you follow the rules

How This Looks

TDirectoryEntry ReturnName()

TStudent

TDirectoryEntry ReturnName()

TFaculty

TGradStudent

TGradStudent

Notes

Resolving Ambiguity Explicitly

- When TDynamicString inherits multiple Print() members

  Make the call explicit by using the class scope operator

  ```
  TDynamicString* aDynamicString;
  ...
  aDynamicString->TString::Print();
  ```

Notes

Although they may be accessed as if they were members of the derived class, inherited members maintain their base class membership

- They can be accessed using the class scope operator

  ```
  aB.A::f("B");
  ```

This is unnecessary except in two instances

- When an inherited member's name is reused (overloaded) in the derived class
  - Reusing an inherited member's name within the derived class hides the inherited member
    - Similar to local identifier reusing name of variable defined at file scope
- When two or more base classes define an inherited member with the same name

Lab 32

- In this lab you will use multiple inheritance to allow polymorphism and resolve ambiguity

Notes

Lab 32 Output

Joe Gard
Grade: B
Advisor: Mr. Sir
Department Comp Sci
Trouble

Joe Inherit
Grade: B
Advisor: Joe Gard
Class of 2000
Trouble

Mr. Sir advises Joe Gard

Joe Gard advises Joe Inherit

Lab Directions

1. Set the directory to Lab 32.
2. Open Student.cp and Student.h
3. Derive `TGraduateStudent` from both `TStudent` and `TFaculty`.
4. Make `aUnderStudent`'s advisor a `TGraduateStudent`.
   (Change `aUnderStudent->AcceptAdvisor(NULL);`)
5. Compile the program.
6. Resolve any ambiguity.
7. Compile and test the program.
8. Make sure your output is exactly as above.

Two Solutions

- Solution 1– Explicit resolution

  We had to initialize both fName's

  aGradStudent->TStudent::AcceptName("Joe Gard");

  aGradStudent->TFaculty::AcceptName("Joe Gard");

  There are some problems:

  A virtual function call, scoped to a class, is invoked as a non-virtual function

  The ambiguity is inherited by subsequent derivation

- Solution 2 – Make it virtual and override it

  A derived member function of the same name hides all instances and provides the necessary functionality

Notes

Enter Virtual Bases

```
class TDirectoryEntry { ...
    char*      ReturnName();
...};

class TFaculty : public virtual TDirectoryEntry {...};
class TStudent: public  virtual TDirectoryEntry {...};
class TGradStudent: public TStudent , public TFaculty{...};

void  TGradStudent::PrintAdvisee() {
    cout << "\n" << this->ReturnName() ...;
...}
```

This now works, but ...

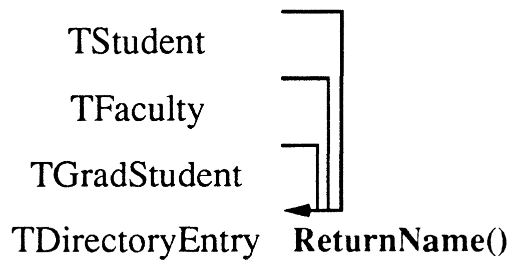
Notes

B and C are derived from A

D has both B and C as base classes

- D will have two A's if A is not a virtual base class, but only one A if A is a virtual base class

How This Looks



Notes

Lab 33

- In this lab you will use a virtual base class to eliminate ambiguity

Notes

Lab 33 Output

Joe Gard
Grade: B
Advisor: Mr. Sir
Department Comp Sci
Trouble

Joe Inherit
Grade: B
Advisor: Joe Gard
Class of 2000
Trouble

Mr. Sir advises Joe Gard

Joe Gard advises Joe Inherit

Lab Directions

1. Set the directory to Lab 33.
2. Open Student.cp and Student.h.
3. Make `TDirectoryEntry` a virtual base for `TFaculty` and `TStudent`.
4. Call `TGradStudent::AcceptName(...)` only once.
5. Compile and test the program.

But

```
void
main() {
    TGradStudent*       aGradStudent = new TGradStudent;
    TDirectoryEntry*    aDirectoryEntry =  aGradStudent;
    aGradStudent = (TGradStudent*)aDirectoryEntry;
    ...
}
```

File "StudentIncM.cp"; line 196 # error:
cast: TDirectoryEntry* ->derived TGradStudent*;
TDirectoryEntry is virtual base

Notes

The Problem With Virtual Bases

- There's no way to convert a pointer to a virtual base back to a pointer to its derived class
- Back to multiple occurrences of a base

```
void TGradStudent::PrintAdvisee() {
    cout << "\n" << this->TStudent::ReturnName() ...;
...}
```

Notes

It doesn't "hurt" to have a base class twice (aside from wasting space because of multiple pointers to the virtual function table, duplicate data, and reduced maintainability)
If you need to cast back from the base class pointer to something else you may not have a choice
If you follow the mixin strategy you are less likely to end up here

Lab 34

- In this lab you examine how casting under multiple inheritance may provide some surprises

Notes

Lab 34 Output

- You will see

Lab Directions

1. Set the directory to Lab 34.
2. Open Student.cp and Student.h
3. Examine the code in `main()` and `TGradStudent::DisplayTGradStudentThis()`, `TDirectoryEntry::DisplayTDirectoryEntryThis()`, and `TFaculty::DisplayTFacultyThis()`
   What does it print?
4. Compile and run the program.
   Are there any surprises ?
5. Change the `TFaculty` and `TStudent` class definitions to make `TDirectoryEntry` a virtual base.
6. Add a new variable of type `TDirectoryEntry*`.
7. Assign `aGradStudent` to that variable.
8. Print out the value of the pointer and call `DisplayTDirectoryEntryThis()` using that pointer.
9. Add a new variable of type `TFaculty*`.
10. Assign `aGradStudent` to that variable.
11. Print out the value of the pointer and call `DisplayTFacultyThis()` using that pointer.
12. Compile and run the program.
13. Explain your results.

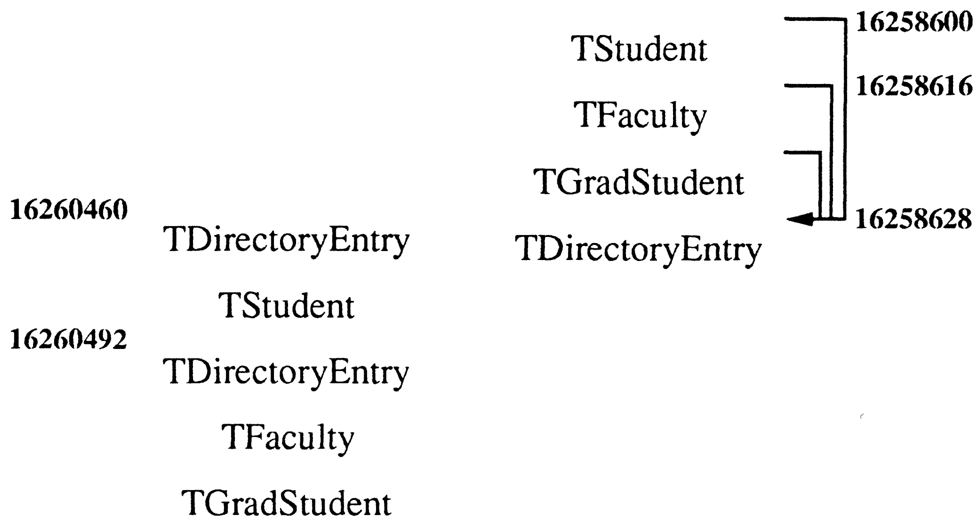
Casting Changes the Pointer!

```
TGradStudent*      aGradStudent = new TGradStudent;
cout << "aGradStudent pointer = " << long(aGradStudent) << "\n";
    aGradStudent pointer = 16258600

TFaculty*      aFaculty;
aFaculty = aGradStudent;
cout << "aFaculty pointer = " << long(aFaculty) << "\n";
    aFaculty pointer = 16258616
```

Notes

Virtual vs Non Virtual Bases



Notes

Non virtual base

```
aGradStudent pointer = 16260460
TGradStudent this pointer = 16260460
TDirectoryEntry this pointer = 16260460
TFaculty this pointer = 16260492
TDirectoryEntry this pointer = 16260492
```

Virtual base

```
aGradStudent pointer = 16258600
TGradStudent this pointer = 16258600
TDirectoryEntry this pointer = 16258628
TFaculty this pointer = 16258616
TDirectoryEntry this pointer = 16258628
aDirectoryEntry pointer = 16258628
TDirectoryEntry this pointer = 16258628
aFaculty pointer = 16258616
TFaculty this pointer = 16258616
```

Multiple Inheritance Issues

- Using HandleObject or PascalObject means you can't use multiple inheritance

    One solution:

        Allocate your own heap zone and manage it yourself

Notes

Using Multiple Inheritance

- Design
- Implementation

Notes

Section 5

Apple Extensions

Notes

This Section's Goals

- Demonstrate competence in:
  - Using Pascal classes from C++
  - Handle-based C++ classes
  - Deriving a C++ class from an Object Pascal class
  - Overriding and adding member functions to a C++ class derived from an Object Pascal class

Notes

Apple Extensions

- `HandleObject`
- `PascalObject`
- `SingleObject`
- Type modifier `pascal`
  Pascal compatible external function declaration
- Optimized `enum`
- Direct function calls – A-traps
- Pascal strings
- SANE interface
- MC68881 and MC68882 coprocessor support

Notes

Keyword pascal
  Provides a Pascal compatible external function declaration
Optimized enum
  Allocates 16 or 32 bits
SingleObject
  Reduces virtual table sizes and overhead
Pascal string support

```
unsigned char*  pasStr = "\pHello";
char*   pasStr "\005Hello";
```

  `&pasStr[0]` is a pointer to a Pascal string
  `&pasStr[1]` is a pointer to a C string
Direct function calls
  Inline machine instructions

Handle-Based Objects

- Handle-based classes

  `class MyClass : public HandleObject { ...`

- Pascal handle-based classes

  `class MyClass : public PascalObject { ...`

  Object Pascal dispatching

  `virtual` and `pascal` allows mixed language hierarchies

  `inherited::`

Notes

Handle-Based objects

CFront generates code that treats pointer as a handle (new and delete use handles instead of pointers)

Declared and used *exactly* as pointer-based objects

Must be created by **`new`** (defined as a pointer): **`THandleClass* aHandleClass = new THandleClass`**

Members accessed as if by pointer: aHandleClass->DoIt();

To derive a handle-based class

`class THandleClass : public HandleObject {...`

Restrictions on handle-based objects

Can be created only by the **`new`** operator

Multiple inheritance cannot be used

Pointers to handle-based classes may be cast only to pointers to handle-based class (or Handle)

Cannot allocate array of handle-based objects

Pascal handle-based objects

Derived from a predefined class – **`PascalObject`**

CFront generates Object Pascal compatible code (and *uses Pascal method dispatching for virtual functions*)

C++ classes can be derived from Pascal classes and methods overriden

`inherited` keyword

Restrictions on pascal handle-based objects

C++ member functions must be **`virtual`** and declared **`pascal`** to be called from Object Pascal

Constructors and destructors allowed ... be careful

Overloading, type conversion functions, and operator functions not allowed for virtual members of pascal classes or functions with pascal attribute

Allowed for other member functions but cannot be declared or accessed from Object Pascal

Handle-Based Object Considerations

- Objects are not locked
- Appear to be pointers

  But are dereferenced handles
- Use local variables

Notes

Using C++ and Object Pascal

- Keyword `pascal`
  - Pascal compatible external function declaration
  - Allows C++ virtual function calls from Object Pascal
- Member functions must be declared
  - `virtual, pascal`
  - derived from `PascalObject`
- A C++ header file
  - Corresponding to the Pascal class and method definitions
- All Pascal base classes must be derived (directly or indirectly) from `PascalObject`

Notes

`pascal`, **`virtual`** and **`PascalObject(TObject)`** allow interface with MacApp

Virtual functions to be defined or referenced from Pascal code must be declared with the pascal keyword

Lab 35

- In this lab you will make a pointer-based C++ class handle-based and write a header file that enables you to use a Pascal class from C++

Notes

Lab 35 Output

```
** Appointment on Monday
    at: 8:30
    with: Neal Goldstein
    about: Class
    for: 8 Hours
    Notes: Bring the teacher an apple (IIfx)
For your appointment on Monday
    with: Neal Goldstein
Bring the teacher an apple (IIfx)
```

Lab Directions

1. Set the directory to Lab 35.
2. Open NoteP.p and NoteP.h.
3. Using the Pascal class definition in NoteP.p, create the corresponding C++ class definition that will allow you to use the `TNoteP` class in your program.
4. Compile the program.
5. Open Schedule.cp and Schedule.h.
6. Modify `TApt` to be derived from a handle-based class.
7. Modify `TApt::PrintAptNote()` to use local variables instead of data members.
8. `HLock` (and `HUnlock`) the `TApt` object when you need to.
9. Compile and test the program.

Note:Schedule.Make file provides for compiling the Pascal unit if needed, and linking in all of the required libraries

Lab 36

- In this lab you will derive a C++ class from a Pascal class, override a Pascal method, and add a new member function

Notes

Lab 36 Output

** Appointment on Monday
at: 8:30
with: Neal Goldstein
about: Class
for: 8 Hours
Notes: Bring the teacher an apple (IIfx)
For your appointment on Monday
with: Neal Goldstein
note : Bring the teacher an apple (IIfx)

Lab Directions

1. Set the directory to Lab 36.
2. Open Schedule.cp and Schedule.h.
3. Create a new class `TNote`, derived form `TNoteP`.
4. Add the member function `INote(char* aNote)`.
5. Override the Pascal method `PrintNote()`.
6. Replace all uses of `TNoteP` with `TNote`.
7. In `TNote::INote(...)` call the Pascal method `INoteP(...)`.
   You will have to convert the C string to a Pascal string (see hint).
8. In the `TNote::PrintNote()` member, add a `cout` statement to reformat the line as shown above, and then call the inherited `Pascal PrintNote()` method.
9. Compile and test the program.

Note:Schedule.Make file provides for compiling the Pascal unit if needed, and linking in all of the required libraries

Section 6

The Future

Notes

C++ Part II

- User defined types – extending the language
    - Operator functions
        - Overloading built in functions (including new and delete
    - User-specified conversions
        - Conversion functions and constructors
    - Memberwise assignment and initialization
        - Initialization constructors
        - Assignment operator overload
- And lots more …

Notes

Future Directions

- Exception handling
- Parameterized types

Notes

Exception Handling

- A standard method for managing exceptions
- Each class currently handles its own exceptions

  This can make it difficult to use large collections and libraries of classes
- The main problem

  The runtime stack must be unwound so that the destructors for class objects that have been unwound are called
- The syntax has been publicly defined

Notes

Parameterized Types or Templates

- Implemented in CFront 3.0
- Functions and classes are dependent on type
- To provide a list class, I have to specify a different class for each list type

```
int*        fIntArray[10];
char*       fCharArray[10];
```

- The same thing is true for functions

```
compare(int, int);
compare(char*, char*);
```

Notes

Parameterized Classes

```
template <class Type>
class TList {
public:
                TList();
    Type&       operator()();
    Type&       operator[] (int);
private:
    Type*       fTypePtr;
    int         fNumber;
};
```

To create this class

```
    TList<TApt>   aList1;
    TList<Ptr>    aList2;
```

Notes

Parameterized Functions

```
template <class Type>
Type&
first(TList<Type>& aList) {
... }

template <class Type>
Type&
TList<Type>::TList() {
... }
```

To define an object of a class:

```
    TList<TApt>   aList1;
    TList<Ptr>    aList2;
```

To invoke a function:

```
    TApt aMyType = first(aList1);
```

Notes

What Next?

Trying to learn object-oriented programming by learning an object-oriented language ...

is like trying to communicate in a foreign language by memorizing a vocabulary list

Notes

Books

Developing Object-Oriented Software for the Macintosh: Analysis, Design, and Programming, by Neal Goldstein and Jeff Alger, Addison-Wesley, 1992.

C++ Primer, by Stanley B. Lippman, 2nd ed., Addison-Wesley, 1991.

Elements of C++ Macintosh Programming, by Dan Weston, Addison-Wesley, 1990.

The Annotated C++ Reference Manual, by Margaret A. Ellis and Bjarne Stroustrup, Addison-Wesley, 1990.

The C++ Programming Language, by Bjarne Stroustrup, 2nd ed., Addison-Wesley, 1991.

Notes

Resources

MacApp$ Group Address
Link "MacApp.Admin" to join

C++$ Group Address
Link "CPlus.Admin" to join

MacApp Developer's Association
Link: MADA

Notes

Magazines

Journal of Object-Oriented Programming
SIGS Publications, Inc.
310 Madison Ave, Suite 503
NY, NY 10017
(212) 972-7055

The C++ Report
310 Madison Avenue, Suite 503
NY, NY 10017
(212) 972-7055

Notes

The power to be your best

Notes