CIS 330:

Lecture 17:
methods implementations &
virtual function table

May 27th, 2015
Hank Childs, University of Oregon
Outline

• Announcements
• Project 3G
• Project 4B
• Review
• Method implementation
• Virtual function table
• Potpourri
Outline

• Announcements
• Project 3G
• Project 4B
• Review
• Method implementation
• Virtual function table
• Potpourri
Announcements

• I need Weds OH to end earlier
• My proposal:
  – Weds 3:25-4:30 in this classroom
Outline

• Announcements
• Project 3G
• Project 4B
• Review
• Method implementation
• Virtual function table
• Potpourri
Project 3G

CIS 330: Project #3G
Assigned: May 25th, 2015
Due June 1st, 2015
(which means submitted by 6am on June 2nd, 2015)
Worth 7% of your grade

*Please read this entire prompt!*

Add 4 new filters:
1) Crop
2) Transpose
3) Invert
4) Checkerboard

Add 1 new source:
1) Constant color

Add 1 new sink:
1) Checksum

Plus: make the two image inputs in Sink be const pointers.
Stress Test Project

• We will have ~60 stress tests
• We can’t check in 60 baseline images and difference them all
  – Will slow ix to a grind
• Solution:
  – We commit “essence of the solution”
  – We also complement that all images posted if needed.
Checksums

Most useful when input is very large and checksum is very small

Input

<table>
<thead>
<tr>
<th>Input</th>
<th>Checksum Function</th>
<th>Checksum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fox</td>
<td>checksum</td>
<td>1582054665</td>
</tr>
<tr>
<td>The red fox jumps over the blue dog</td>
<td>checksum</td>
<td>2367213558</td>
</tr>
<tr>
<td>The red fox jumps over the blue dog</td>
<td>checksum</td>
<td>3043859473</td>
</tr>
<tr>
<td>The red fox jumps over the blue dog</td>
<td>checksum</td>
<td>1321115126</td>
</tr>
<tr>
<td>The red fox jumps over the blue dog</td>
<td>checksum</td>
<td>1685473544</td>
</tr>
</tbody>
</table>

From Wikipedia
Our “checksum”

• Three integers:
  – Sum of red channel
  – Sum of green channel
  – Sum of blue channel

• When you create a stress test, you register these three integers

• When you test against others stress tests, you compare against their integers
  – If they match, you assume you got it right
  – Otherwise, you check their webpage

This will be done with a derived type of Sink.
Outline

• Announcements
• Project 3G
• Project 4B
• Review
• Method implementation
• Virtual function table
• Potpourri
Project 4B

- Run valgrind on your 3E project.
- Repeat until:
  - no memory errors
  - no memory leaks
Project 4B

• What to turn in:
  – A screen shot of valgrind showing no errors

```bash
hank@ix: ~/3F/3F 65$ valgrind proj3F ~/3A_input.pnm 3F_output.pnm
==16125== Memcheck, a memory error detector
==16125== Copyright (C) 2002-2011, and GNU GPL'd, by Julian Seward et al.
==16125== Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright info
==16125== Command: proj3F /home/users/hank/3A_input.pnm 3F_output.pnm
==16125==
==16125== HEAP SUMMARY:
==16125==    in use at exit: 0 bytes in 0 blocks
==16125==    total heap usage: 33 allocs, 33 frees, 108,022,422 bytes allocated
==16125== All heap blocks were freed -- no leaks are possible
==16125==
==16125== For counts of detected and suppressed errors, rerun with: -v
==16125== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 2 from 2)
```
Outline

• Announcements
• Project 3G
• Project 4B
• Review
• Method implementation
• Virtual function table
• Potpourri
globals

• You can create global variables that exist outside functions.

```c
#include <stdio.h>
int X = 5;

int main()
{
    printf("X is %d\n", X);
}
```

```bash
fawcett:Documents child$ cat global1.C
fawcett:Documents child$ g++ global1.C
fawcett:Documents child$ ./a.out
X is 5
fawcett:Documents child$ 
```
global variables

- global variables are initialized before you enter main
Storage of global variables...

- Global variables are stored in a special part of memory — “data segment” (not heap, not stack)
- If you re-use global names, you can have collisions

```c
fawcett:Documents childs$ cat file1.C
int X = 6;

int main()
{
}

fawcett:Documents childs$ g++ -c file1.C

fawcett:Documents childs$ cat file2.C
int X = 7;

int doubler(int Y)
{
    return 2*Y;
}

fawcett:Documents childs$ g++ -c file2.C

fawcett:Documents childs$ g++ file1.o file2.o
ld: duplicate symbol _X in file2.o and file1.o
collect2: ld returned 1 exit status
```
Externs: mechanism for unifying global variables across multiple files

```c
fawcett:330 childs$ cat file1.C
#include <stdio.h>
int count = 0;
int doubler(int);
int main()
{
    count++;
    doubler(3);
    printf("count is %d\n", count);
}
```

```bash
fawcett:330 childs$ cat file2.C
extern int count;
int doubler(int Y)
{
    count++;
    return 2*Y;
}

fawcett:330 childs$ g++ -c file1.C
fawcett:330 childs$ g++ -c file2.C
fawcett:330 childs$ g++ file1.o file2.o
fawcett:330 childs$ ./a.out
count is 2
```

extern: there’s a global variable, and it lives in a different file.
static

• static memory: third kind of memory allocation
  – reserved at compile time
• contrasts with dynamic (heap) and automatic (stack) memory allocations
• accomplished via keyword that modifies variables

There are three distinct usages of statics
static usage #1: persistency within a function

```c
fawcett:330 childds$ cat static1.C
#include <stdio.h>

int fibonacci()
{
    static int last2 = 0;
    static int last1 = 1;
    int rv = last1+last2;
    last2 = last1;
    last1 = rv;
    return rv;
}

int main()
{
    int i;
    for (int i = 0 ; i < 10 ; i++)
        printf("%d\n", fibonacci());
}
```

static usage #2: making global variables be local to a file

I have no idea why the static keyword is used in this way.
static usage #3: making a singleton for a class

```cpp
fawcett:Downloads childs$ cat static3.C
#include <iostream>

using std::cout;
using std::endl;

class MyClass
{
  public:
    MyClass() { numInstances++; }
    virtual ~MyClass() { numInstances--; }

    int GetNumInstances() { return numInstances; }

  private:
    int numInstances;
};

int main()
{
    MyClass *p = new MyClass[10];
    cout << "Num instances = " << p[0].GetNumInstances() << endl;
    delete [] p;
    cout << "Num instances = " << p[0].GetNumInstances() << endl;
}
```

fawcett:Downloads childs$ g++ static3.C
fawcett:Downloads childs$ ./a.out
Num instances = 1
Num instances = 0
fawcett:Downloads childs$
static usage #3: making a singleton for a class

```
#include <iostream>

using std::cout;
using std::endl;

class MyClass
{
  public:
    MyClass();
    virtual ~MyClass();

    int GetNumInstances();

  private:
    static int numInstances;
};

int main()
{
  delete[] p;
  cout << "Num instances = " << p[0].GetNumInstances() << endl;
}
```

We have to tell the compiler where to store this static.

What do we get?
static usage #3: making a singleton for a class

```cpp
fawcett:Downloads childs$ cat static3.C
#include <iostream>

using std::cout;
using std::endl;

class MyClass
{
    public:
        MyClass() { numInstances++; };
        virtual ~MyClass() { numInstances--; };

        int GetNumInstances(void) { return numInstances; };

    private:
        static int numInstances;
};

int MyClass::numInstances = 0;

int main()
{
    MyClass *p = new MyClass[10];
    cout << "Num instances = " << p[0].GetNumInstances() << endl;
    delete [] p;
    cout << "Num instances = " << p[0].GetNumInstances() << endl;
}
static methods

Static data members and static methods are useful and they are definitely used in practice.
int X = 0;

class MyClass
{
  public:
    MyClass() { X = 1; };

    void SetValue(int);

  private:
    int X;
};

void MyClass::SetValue(int X)
{
  int X = 3;
  cout << "X is " << X << endl;
}

int main()
{
  MyClass mc;
  mc.SetValue(2);
}
int X = 0;

class MyClass
{
    public:
        MyClass() { X = 1; };

    void SetValue(int);

    private:
        int X;
};

void MyClass::SetValue(int X)
{
    {
        int X = 3;
        cout << "X is " << X << endl;
    }
}

int main()
{
    MyClass mc;
    mc.SetValue(2);
}
int X = 0;

class MyClass {
    public:
        MyClass() { X = 1; };

        void SetValue(int);

    private:
        int  X;
};

void MyClass::SetValue(int X) {
    {
        int X = 3;
        cout << "X is " << X << endl;
    }
}

int main()
{
    MyClass mc;
    mc.SetValue(2);
}
```cpp
int X = 0;

class MyClass
{
    public:
        MyClass() { X = 1; };

    void SetValue(int);

    private:
        int X;
};

void MyClass::SetValue(int X)
{
    int X = 3;
    cout << "X is " << X << endl;
}

int main()
{
    MyClass mc;
    mc.SetValue(2);
}
```

What does this one print?

Answer: 1
int X = 0;

class MyClass
{
  public:
    MyClass() { X = 1; };

    void SetValue(int);

  private:
    int X;
};

void MyClass::SetValue(int x)
{
  int X = 3;
  cout << "X is " << X << endl;
}

int main()
{
  MyClass mc;
  mc.SetValue(2);
}
Scope Rules

• The compiler looks for variables:
  – inside a function or block
  – function arguments
  – data members (methods only)
  – globals
Shadowing

- Shadowing is a term used to describe a “subtle” scope issue.
  - ... i.e., you have created a situation where it is confusing which variable you are referring to

```cpp
class Sink
{
    public:
        void SetInput(Image *i) { input = i; }
    protected:
        Image *input;
};

class Writer : public Sink
{
    public:
        void Write(void) { /* write input */ }
    protected:
        Image *input;
};

int main()
{
    Writer writer;
    writer.SetInput(image);
    writer.Write();
}
```
(This is now new content I didn’t make it to last week)
C++ lets you define operators

• You declare a method that uses an operator in conjunction with a class
  – +, -, /, !, ++, etc.

• You can then use operator in your code, since the compiler now understands how to use the operator with your class

• This is called “operator overloading”
  – ... we are overloading the use of the operator for more than just the simple types.
Example of operator overloading

class MyInt
{
    public:
        MyInt(int x) { myInt = x; }
    MyInt& operator++();
    int GetValue(void) { return myInt; }

    protected:
        int myInt;
};

MyInt &
MyInt::operator++()
{
    myInt++;
    return *this;
}

int main()
{
    MyInt mi(6);
    ++mi;
    ++mi;
    printf("Value is %d\n", mi.GetValue());
}
fawcett:330 childs$ ./a.out
Value is 8

Define operator ++ for MyInt

Declare operator ++ will be overloaded for MyInt

Call operator ++ on MyInt.
More operator overloading

```cpp
#include <iostream>

using std::ostream;
using std::cout;
using std::endl;

class Image
{
    public:
        Image();

        friend ostream& operator<<(ostream &os, const Image &);

    private:
        int width, height;
        unsigned char *buffer;
};

Image::Image()
{
    width = 100;
    height = 100;
    buffer = NULL;
}

ostream &
operator<<(ostream &out, const Image &img)
{
    out << img.width << "x" << img.height << endl;
    if (img.buffer == NULL)
        out << "No buffer allocated!" << endl;
    else
        out << "Buffer is allocated!" << endl;
}

int main()
{
    Image img;
    cout << img;
}
```

```bash
fawcett:330 childs$ g++ oostream.C
fawcett:330 childs$ ./a.out
100x100
No buffer allocated!
```
Beauty of inheritance

• ostream provides an abstraction
  – That’s all Image needs to know
    • it is a stream that is an output
  – You code to that interface
  – All ostream’s work with it

```c
int main()
{
    Image img;
    cerr << img;
}
fawcett:330 childds$ ./a.out
100x100
No buffer allocated!
```
```c
int main()
{
    Image img;
    ofstream ofile("output_file");
    ofile << img;
}
fawcett:330 childds$ g++ oostream.C
fawcett:330 childds$ ./a.out
fawcett:330 childds$ cat output_file
100x100
No buffer allocated!
```
class Image
{
    public:
        Image();
        SetSize(int w, int h);

    friend ostream& operator<<(ostream &os, const Image &);

    Image & operator=(const Image &);
};

void Image::SetSize(int w, int h)
{
    if (buffer != NULL)
        delete [] buffer;
    width = w;
    height = h;
    buffer = new unsigned char[3*width*height];
}

Image &
Image::operator=(const Image &rhs)
{
    if (buffer != NULL)
        delete [] buffer;
    buffer = NULL;

    width = rhs.width;
    height = rhs.height;
    if (rhs.buffer != NULL)
    {
        buffer = new unsigned char[3*width*height];
        memcpy(buffer, rhs.buffer, 3*width*height);
    }

    int main()
    {
        Image img1, img2;
        img1.SetSize(200, 200);
        cout << "Image 1:" << img1;
        cout << "Image 2:" << img2;
        img2 = img1;
        cout << "Image 1:" << img1;
        cout << "Image 2:" << img2;
    }
let’s do this again...

```cpp
ostream &
operator<<(ostream &out, const Image &img) {
    out << img.width << "x" << img.height << endl;
    if (img.buffer == NULL)
        out << "No buffer allocated!" << endl;
    else
        out << "Buffer is allocated, and value is "
            << (void *) img.buffer << endl;

    return out;
}
```

```
fawcett:330 childs$ ./a.out
Image 1:200x200
Buffer is allocated, and value is 0x100800000
Image 2:0x0
No buffer allocated!
Image 1:200x200
Buffer is allocated, and value is 0x100800000
Image 2:200x200
Buffer is allocated, and value is 0x10081e600
```
let’s do this again...

class Image
{
  public:
     Image();
     void SetSize(int w, int h);
     friend ostream& operator<<(ostream &os, const Image &);
  // Image & operator=(const Image &);
  private:
    int width, height;
    unsigned char *buffer;
};

int main()
{
  Image img1, img2;
  img1.SetSize(200, 200);
  cout << "Image 1:" << img1;
  cout << "Image 2:" << img2;
  img2 = img1;
  cout << "Image 1:" << img1;
  cout << "Image 2:" << img2;
}

fawcett:330 childs$ g++ assignment_op.C
fawcett:330 childs$  

it still compiled ... why?
C++ defines a default assignment operator for you

- This assignment operator does a bitwise copy from one object to the other.
- Does anyone see a problem with this?

```
fawcett:330 childs$ ./a.out
Image 1:200x200
Buffer is allocated, and value is 0x10080000
Image 2:0x0
No buffer allocated!
Image 1:200x200
Buffer is allocated, and value is 0x10080000
Image 2:200x200
Buffer is allocated, and value is 0x10080000
```

This behavior is sometimes OK and sometimes disastrous.
Copy constructors: same deal

• C++ automatically defines a copy constructor that does bitwise copying.

• Solutions for copy constructor and assignment operators:
  – Re-define them yourself to do “the right thing”
  – Re-define them yourself to throw exceptions
  – Make them private so they can’t be called
Outline

• Announcements
• Project 3G
• Project 4B
• Review
  • Method implementation
  • Virtual function table
• Potpourri
“this”: pointer to current object

- From within any struct’s method, you can refer to the current object using “this”
How methods work under the covers (1/4)

class MyIntClass
{
    public:
        MyIntClass(int x) { myInt = x; }
        friend void FriendIncrementFunction(MyIntClass *);
        int GetMyInt() { return myInt; }
    protected:
        int myInt;
};

void FriendIncrementFunction(MyIntClass *mic)
{
    mic->myInt++;
}

int main()
{
    MyIntClass MIC(12);
    FriendIncrementFunction(&MIC);
    FriendIncrementFunction(&MIC);
    cout << "My int is " << MIC.GetMyInt() << endl;
}
How methods work under the covers (2/4)

class MyIntClass
{
    public:
    MyIntClass(int x) { myInt = x; }

    friend void FriendIncrementFunction(MyIntClass *mic);
    int GetMyInt() { return myInt; }

    protected:
    int myInt;
};

void FriendIncrementFunction(MyIntClass *mic)
{
    mic->myInt++;
}

int main()
{
    MyIntClass MIC(12);
    FriendIncrementFunction(&MIC);
    FriendIncrementFunction(&MIC);
    cout << "My int is " << MIC.GetMyInt() << endl;
}
How methods work under the covers (3/4)

class MyIntClass
{
    public:
        MyIntClass(int x) { myInt = x; };

        friend void FriendIncrementFunction(MyIntClass *);

        void IncrementMethod(void);

        int GetMyInt() { return myInt; };

    protected:
        int myInt;
};

void FriendIncrementFunction(MyIntClass *mic)
{
    mic->myInt++;
}

void MyIntClass::IncrementMethod(void)
{
    this->myInt++;
}

int main()
{
    MyIntClass MIC(12);
    FriendIncrementFunction(&MIC);
    MIC.IncrementMethod();
    cout << "My int is " << MIC.GetMyInt() << endl;
}
How methods work under the covers (4/4)

The compiler secretly slips “this” onto the stack whenever you make a method call.

It also automatically changes “myInt” to this->myInt in methods.

```cpp
class MyIntClass {
    int myInt;

public:
    MyIntClass(int i) { this->myInt = i; }
    void increment() { this->myInt++; }
};

void MyIntClass::IncrementMethod(void) {
    this->myInt++;  // Increment this->myInt
}

int main() {
    MyIntClass MIC(12);
    MIC.IncrementMethod();
    cout << "My int is " << MIC.GetMyInt() << endl;
}
```
Outline

• Announcements
• Project 3G
• Project 4B
• Review
• Method implementation
• Virtual function table
• Potpourri
Virtual functions

- Virtual function: function defined in the base type, but can be re-defined in derived type.
- When you call a virtual function, you get the version defined by the derived type
```c
#include <stdio.h>

struct SimpleID {
    int id;
    virtual int GetIdentifier() { return id; };
};

struct ComplexID : SimpleID {
    int extraId;
    virtual int GetIdentifier() { return extraId*128+id; };
};

int main() {
    ComplexID cid;
    cid.id = 3;
    cid.extraId = 3;
    printf("ID = %d\n", cid.GetIdentifier());
}
```

Picking the right virtual function

```cpp
class A
{
  public:
    virtual const char *GetType() { return "A"; }
};

class B : public A
{
  public:
    virtual const char *GetType() { return "B"; }
};

int main()
{
  A a;
  B b;

  cout << "a is " << a.GetType() << endl;
  cout << "b is " << b.GetType() << endl;
}
```

It seems like the compiler should be able to figure this out ... it knows that a is of type A and it knows that b is of type B

```bash
fawcett:330 childs$ g++ virtual.C
fawcett:330 childs$ .a.out
??????
```
Picking the right virtual function

class A
{
    public:
    virtual const char *GetType() { return "A"; }
};

class B : public A
{
    public:
    virtual const char *GetType() { return "B"; }
};

void ClassPrinter(A *ptrToA)
{
    cout << "ptr points to a " << ptrToA->GetType() << endl;
}

int main()
{
    A a;
    B b;

    ClassPrinter(&a);
    ClassPrinter(&b);
}

So how does the compiler know?

How does it get “B” for “b” and “A” for “a”?
Virtual Function Table

• Let C be a class and X be an instance of C.
• Let C have 3 virtual functions & 4 non-virtual functions
• C has a hidden data member called the “virtual function table”
• This table has 3 rows
  – Each row has the correct definition of the virtual function to call for a “C”.
• When you call a virtual function, this table is consulted to locate the correct definition.
Showing the existence of the virtual function pointer with sizeof()

empty objects have size of 1? why?!?

Answer: so every object has a unique address.

```cpp
class A {
  public:
    virtual
};
class B : public A {
  public:
    virtual
};
class C {
  public:
    const char *Get_Type() { return "C"; }
};
int main()
{
  A a;
  B b;
  cout << "Size of A is " << sizeof(A) << endl;
  cout << "Size of a pointer is " << sizeof(int *) << endl;
  cout << "Size of C is " << sizeof(C) << endl;
}
```

what will this print?

```bash
fawcett:330 childsls$ ./a.out
Size of A is 8
Size of a pointer is 8
Size of C is 1
```
Virtual Function Table

• Let C be a class and X be an instance of C.
• Let C have 3 virtual functions & 4 non-virtual functions
• Let D be a class that inherits from C and Y be an instance of D.
  – Let D add a new virtual function
• D’s virtual function table has 4 rows
  – Each row has the correct definition of the virtual function to call for a “D”.


More notes on virtual function tables

• There is one instance of a virtual function table for each class
  – Each instance of a class shares the same virtual function table

• Easy to overwrite (i.e., with a memory error)
  – And then all your virtual function calls will be corrected
  – Don’t do this! ;)

Virtual function table: example

CIS 330: Project #2C
Assigned: April 17th, 2014
Due April 24th, 2014
(which means submitted by 6am on April 25th, 2014)
Worth 6% of your grade

Please read this entire prompt!

Assignment: You will implement subtypes with C.

1) Make a union called ShapeUnion with the three types (Circle, Rectangle, Triangle).
2) Make a struct called FunctionTable that has pointers to functions.
3) Make an enum called ShapeType that identifies the three types.
4) Make a struct called Shape that has a ShapeUnion, a ShapeType, and a FunctionTable.
5) Modify your 9 functions to deal with Shapes.
6) Integrate with the new driver function. Test that it produces the correct output.
Virtual function table: example

class Shape
{
    virtual double GetArea() = 0;
    virtual void GetBoundingBox(double *) = 0;
};

class Rectangle : public Shape
{
    public:
        Rectangle(double, double, double, double, double);
    virtual double GetArea();
    virtual void GetBoundingBox(double *);
    protected:
        double minX, maxX, minY, maxY;
};

class Triangle : public Shape
{
    public:
        Triangle(double, double, double, double, double);
    virtual double GetArea();
    virtual void GetBoundingBox(double *);
    protected:
        double pt1X, pt2X, minY, maxY;
};
Questions

• What does the virtual function table look like for a Shape?

```
typedef struct
{
    double (*GetArea)(Shape *);
    void  (*GetBoundingBox)(Shape *, double *);
} VirtualFunctionTable;
```

• What does Shape’s virtual function table look like?
  – Trick question: Shape can’t be instantiated, precisely because you can’t make a virtual function table
  • abstract type due to pure virtual functions
Questions

• What is the virtual function table for Rectangle?

```c
  c->ft.GetArea = GetRectangleArea;
  c->ft.GetBoundingBox = GetRectangleBoundingBox;
```

• (this is a code fragment from my 2C solution)
Calling a virtual function

• Let X be an instance of class C.
• Let the virtual function be the 4\(^{th}\) function.
• Let the arguments to the virtual function be an integer Y and a float Z.
• Then call:
  
  \((X.vptr[3])(\&X, Y, Z);\)

  The pointer to the virtual function pointer (often called a vptr) is a data member of X.

  The 4\(^{th}\) virtual function has index 3 (0-indexing).

  Secretly pass “this” as first argument to method.
This whole scheme gets much harder with virtual inheritance, and you have to carry around multiple virtual function tables. 

```cpp
class A
{
public:
    virtual void Foo2();
};

class C
{
public:
    virtual void Foo1();
    virtual void Foo2();
    virtual void Foo3();
};
```

Same as B’s

This is how you can treat a C as a B
Virtual Function Table: Summary

• Virtual functions require machinery to ensure the correct form of a virtual function is called
• This is implemented through a virtual function table
• Every instance of a class that has virtual functions has a pointer to its class’s virtual function table
• The virtual function is called via following pointers
  – Performance issue
Outline

• Announcements
• Project 3G
• Project 4B
• Review
• Method implementation
• Virtual function table
• Potpourri
Upcasting and Downcasting

• Upcast: treat an object as the base type
  – We do this all the time!
  – Treat a Rectangle as a Shape

• Downcast: treat a base type as its derived type
  – We don’t do this one often
  – Treat a Shape as a Rectangle
    • You better know that Shape really is a Rectangle!!
Upcasting and Downcasting

class A {
};

class B : public A {
    public:
        B() { myInt = 5; };
    void Printer(void) { cout << myInt << endl; };

    private:
        int myInt;
};

void Downcaster(A *a) {
    B *b = (B *) a;
    b->Printer();
}

int main() {
    A a;
    B b;

    Downcaster(&b); // no problem
    Downcaster(&a); // no good
}

what do we get?

fawcett:330 childs$ g++ downcaster.C
fawcett:330 childs$ ./a.out
5
-1074118656
Upcasting and Downcasting

- C++ has a built in facility to assist with downcasting: `dynamic_cast`
- I personally haven’t used it a lot, but it is used in practice
- Ties in to std::exception
Default Arguments

```cpp
void Foo(int X, int Y = 2)
{
    cout << "X = " << X << ", Y = " << Y << endl;
}

int main()
{
    Foo(5);
    Foo(5, 4);
}
```

default arguments: compiler pushes values on the stack for you if you choose not to enter them.
Booleans

• New simple data type: bool (Boolean)
• New keywords: true and false

```c
int main()
{
    bool b = true;
    cout << "Size of boolean is " << sizeof(bool) << endl;
}
```

```
fawcett:330 childs$ g++ Boolean.C
fawcett:330 childs$ ./a.out
```
Inline function

• inlined functions:
  – hint to a compiler that can improve performance
  – basic idea: don’t actually make this be a separate function that is called
    • Instead, just pull the code out of it and place it inside the current function
  – new keyword: inline

```c
inline int doubler(int X)
{
    return 2*X;
}

int main()
{
    int Y = 4;
    int Z = doubler(Y);
}
```

The compiler sometimes refuses your inline request (when it thinks inlining won’t improve performance), but it does it silently.
Inlines can be automatically done within class definitions

- Even though you don’t declare this as inline, the compiler treats it as an inline

```java
class MyDoublerClass {
    int doubler(int X) { return 2*X; };
};
```
You should only do inlines within header files

Left: function is inlined in every .C that includes it ... no problem
Right: function is defined in every .C that includes it ... duplicate symbols
Backup slides
Backgrounding

• “&”: tell shell to run a job in the background
  – Background means that the shell acts as normal, but the command you invoke is running at the same time.

• “sleep 60” vs “sleep 60 &”

When would backgrounding be useful?
Suspending Jobs

• You can suspend a job that is running
  Press “Ctrl-Z”
• The OS will then stop job from running and not schedule it to run.
• You can then:
  – make the job run in the background.
    • Type “bg”
  – make the job run in the foreground.
    • Type “fg”
      – like you never suspended it at all!!