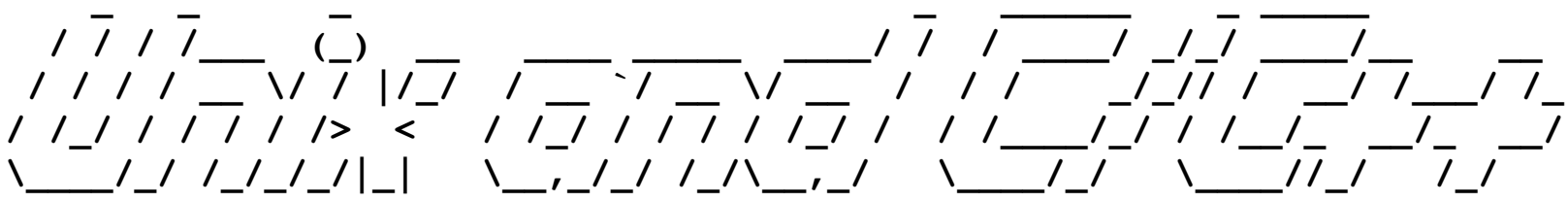
UNIVERSITY OF OREGON

CIS 330:



Lecture 14: more class, C++ streams

May 9th, 2018

Hank Childs, University of Oregon

Announcements

- Projects
  - 3B assigned Friday, due today
  - 3C posted today, due May 16
  - 3D posted today, also due May 16
    - 3D is not required to do 3E, etc.
    - So you can skip it, although you will lose points.
  - OH:
    - Weds 12-1 (in Hank's office)
    - Thurs 4-5 (in Hank's office)

Schedule rest of this week

- Friday: Lab 4 (Brent)
- Friday OH: with Brent, not Hank
  - Brent will announce location (125 MCK or 100 DES)

Project 3B

- Retrofit to use references
- Add useful routines for manipulating an image
  - Halve in size
  - Concatenate
  - Crop
  - Blend

Review

Simple inheritance example

```
struct A
{
    int x;
};

struct B : A
{
    int y;
};

int main()
{
    B b;
    b.x = 3;
    b.y = 4;
}
```

- Terminology
  - B inherits from A
  - A is a base type for B
  - B is a derived type of A
- Noteworthy
  - ":" (during struct definition) → inherits from
    - Everything from A is accessible in B
      - (b.x is valid!!)

Object sizes

```
128-223-223-72-wireless:330 hank$ cat simple_inheritance.C
#include <stdio.h>

struct A
{
    int x;
};

struct B : A
{
    int y;
};

int main()
{
    B b;
    b.x = 3;
    b.y = 4;
    printf("Size of A = %lu, size of B = %lu\n", sizeof(A), sizeof(B));
}
128-223-223-72-wireless:330 hank$ g++ simple_inheritance.C
128-223-223-72-wireless:330 hank$ ./a.out
Size of A = 4, size of B = 8
```

Inheritance + TallyCounter

```
struct TallyCounter
{
    friend   int main();

  public:
           TallyCounter(void);
           TallyCounter(int c);
           TallyCounter(TallyCounter &);

  private:
    int    count;

  public:
    void   Reset();
    int    GetCount();
    void   IncrementCount();
};

struct FancyTallyCounter : TallyCounter
{
    void   DecrementCount() { count--; }
}
```

FancyTallyCounter inherits all of TallyCounter, and adds a new method: DecrementCount

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

Virtual functions: example

```
128-223-223-72-wireless:330 hank$ cat virtual.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());
}
128-223-223-72-wireless:330 hank$ g++ virtual.C
128-223-223-72-wireless:330 hank$ ./a.out
ID = 387
```

Virtual functions: example

```
128-223-223-72-wireless:330 hank$ cat virtual2.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; };
};

struct C3 : ComplexID
{
   int extraExtraId;
};

int main()
{
   C3 cid;
   cid.id = 3;
   cid.extraId = 3;
   cid.extraExtraId = 4;
   printf("ID = %d\n", cid.GetIdentifier());
}
128-223-223-72-wireless:330 hank$ g++ virtual2.C
128-223-223-72-wireless:330 hank$ ./a.out
```

You get the method furthest down in the inheritance hierarchy

public / private inheritance

- class A : [public|private] B
  - → class A : public B
  - → class A : private B

- So:
  - For public, base class's public members will be public
  - For private, base class's public members will be private

- Public common
  - I've never personally used anything else

NEW SLIDE

- public inheritance → no restriction beyond what restrictions in base class
  - Example:
    - class A { private: int x; }; class B : public A {};
    - → B cannot access x
- private inheritance → *does* restrict beyond what restrictions in base class
  - Example 2:
    - class A { public: int x; }; class B : private A {};
    - → B again cannot access x

public / private inheritance

- class A : public B
  - A “is a” B
- class A : private B
  - A “is implemented using” B
    - And: !(A “is a” B)
    - … you can’t treat A as a B

One more access control word: protected

- Protected means:
  - It cannot be accessed outside the object
    - Modulo “friend”
  - But it can be accessed by derived types
    - (assuming public inheritance)

Memory Management

C++ memory management

- C++ provides new constructs for requesting heap memory from the memory manager
  - stack memory management is not changed
    - (automatic before, automatic now)
- Allocate memory: “new”
- Deallocate memory: “delete”

new / delete syntax

No header necessary

```
fawcett:330 childs$ cat new.C
int main()
{
    int *oneInt = new int;
    *oneInt = 3;
    int *intArray = new int[3];
    intArray[0] = intArray[1] = intArray[2] = 5;

    delete oneInt;
    delete [] intArray;
}
```

Allocating array and single value is the same.

Deleting array takes [], deleting single value doesn't.

new knows the type and allocates the right amount.

new int → 4 bytes
new int[3] → 12 bytes

new calls constructors for your classes

- Declare variable in the stack: constructor called
- Declare variable with “malloc”: constructor not called
  - C knows nothing about C++!
- Declare variable with “new”: constructor called

More on Classes

Destructors

- A destructor is called automatically when an object goes out of scope (via stack or delete)
- A destructor's job is to clean up before the object disappears
  - Deleting memory
  - Other cleanup (e.g., linked lists)
- Same naming convention as a constructor, but with a prepended ~ (tilde)

Destructors example

```
struct Pixel
{
    unsigned char R, G, B;
};

class Image
{
   public:
        Image(int w, int h);
       ~Image();

   private:
        int width, height;
        Pixel *buffer;
};

Image::Image(int w, int h)
{
    width = w; height = h;
    buffer = new Pixel[width*height];
}

Image::~Image()
{
    delete [] buffer;
}
```

Class name with ~ prepended

Defined like any other method, does cleanup

If Pixel had a constructor or destructor, it would be getting called (a bunch) by the new's and delete's.

Inheritance and Constructors/Destructors: Example

- Constructors from base class called first, then next derived type second, and so on.
- Destructor from base class called last, then next derived type second to last, and so on.

- Derived type always assumes base class exists and is set up
  - … base class never needs to know anything about derived types

Inheritance and Constructors/Destructors: Example

```
#include <stdio.h>

class C
{
  public:
    C()  { printf("Constructing C\n"); };
   ~C()  { printf("Destructing C\n"); };
};

class D : public C
{
  public:
    D()  { printf("Constructing D\n"); };
   ~D()  { printf("Destructing D\n"); };
};

int main()
{
    printf("Making a D\n");
    {
        D b;
    }

    printf("Making another D\n");
    {
        D b;
    }
}
```

```
Making a D
Constructing C
Constructing D
Destructing D
Destructing C
Making another D
Constructing C
Constructing D
Destructing D
Destructing C
```

Possible to get the wrong destructor

- With a constructor, you always know what type you are constructing.
- With a destructor, you don't always know what type you are destructing.
- This can sometimes lead to the wrong destructor getting called.

Getting the wrong destructor

```
#include <stdio.h>

class C
{
  public:
    C()  { printf("Constructing C\n"); };
   ~C()  { printf("Destructing C\n"); };
};

class D : public C
{
  public:
    D()  { printf("Constructing D\n"); };
   ~D()  { printf("Destructing D\n"); };
};

D* D_as_D_Creator() { return new D; };
C* D_as_C_Creator() { return new D; };

int main()
{
    C* c = D_as_C_Creator();
    D* d = D_as_D_Creator();

    delete c;
    delete d;
}
```

```
fawcett:330 childs$ ./a.out
Constructing C
Constructing D
Constructing C
Constructing D
Destructing C
Destructing D
Destructing C
```

Virtual destructors

- Solution to this problem:

  Make the destructor be declared virtual

- Then existing infrastructure will solve the problem
  - … this is what virtual functions do!

Virtual destructors

```
#include <stdio.h>

class C
{
  public:
    C()  { printf("Constructing C\n"); };
   virtual ~C()  { printf("Destructing C\n"); };
};

class D : public C
{
  public:
    D()  { printf("Constructing D\n"); };
   virtual ~D()  { printf("Destructing D\n"); };
};

D* D_as_D_Creator() { return new D; };
C* D_as_C_Creator() { return new D; };

int main()
{
    C* c = D_as_C_Creator();
    D* d = D_as_D_Creator();

    delete c;
    delete d;
}
```

```
fawcett:330 childs$ ./a.out
Constructing C
Constructing D
Constructing C
Constructing D
Destructing D
Destructing C
Destructing D
Destructing C
```

NEW STUFF

Objects in objects

```
#include <stdio.h>

class A
{
  public:
    A()  { printf("Constructing A\n"); };
   ~A()  { printf("Destructing A\n"); };
};

class B
{
  public:
    B()  { printf("Constructing B\n"); };
   ~B()  { printf("Destructing B\n"); };
  private:
    A a1, a2;
};

int main()
{
    printf("Making a B\n");
    {
        B b;
    }

    printf("Making another B\n");
    {
        B b;
    }
}
```

By the time you enter B's constructor, a1 and a2 are already valid.

```
ut
Destructing A
Destructing A
Making another B
Constructing A
Constructing A
Constructing B
Destructing B
Destructing A
Destructing A
```

Objects in objects

```
#include <stdio.h>

class A
{
  public:
    A()  { printf("Constructing A\n"); };
   ~A()  { printf("Destructing A\n"); };
};

class B
{
  public:
    B()  { printf("Constructing B\n"); };
   ~B()  { printf("Destructing B\n"); };
};

class C
{
  public:
    C()  { printf("Constructing C\n"); };
   ~C()  { printf("Destructing C\n"); };
  private:
    A  a;
    B  b;
};

int main()
{
    C c;
}
```

```
fawcett:330 childs$ ./a.out
Constructing A
Constructing B
Constructing C
Destructing C
Destructing B
Destructing A
```

Objects in objects: order is important

```
#include <stdio.h>

class A
{
  public:
    A()  { printf("Constructing A\n"); };
   ~A()  { printf("Destructing A\n"); };
};

class B
{
  public:
    B()  { printf("Constructing B\n"); };
   ~B()  { printf("Destructing B\n"); };
};

class C
{
  public:
    C()  { printf("Constructing C\n"); };
   ~C()  { printf("Destructing C\n"); };
  private:
    B  b;
    A  a;
};

int main()
{
    C c;
}
```

```
fawcett:330 childs$ ./a.out
Constructing B
Constructing A
Constructing C
Destructing C
Destructing A
Destructing B
```

Initializers

- New syntax to have variables initialized before even entering the constructor

```
#include <stdio.h>

class A
{
  public:
    A() : x(5)
    {
        printf("x is %d\n", x);
    };
  private:
    int x;
};

int main()
{
    A a;
}
```

```
fawcett:330 childs$ ./a.out
x is 5
```

Initializers

- Initializers are a mechanism to have a constructor pass arguments to another constructor
- Needed because
  - Base class constructors are called before derived constructors & need to pass arguments in derived constructor to base class
  - Constructors for objects contained in a class are called before the container class & need to pass arguments in container class's destructor

Initializers

- Needed because
  - Constructors for objects contained in a class are called before the container class & need to pass arguments in container class's destructor

```
#include <stdio.h>

class A
{
  public:
    A(int x)  { v = x; };
  private:
    int v;
};

class B
{
  public:
    B(int x)  { v = x; };
  private:
    int v;
};

class C
{
  public:
    C(int x, int y) : b(x), a(y)  { };
  private:
    B  b;
    A  a;
};

int main()
{
    C c(3,5);
}
```

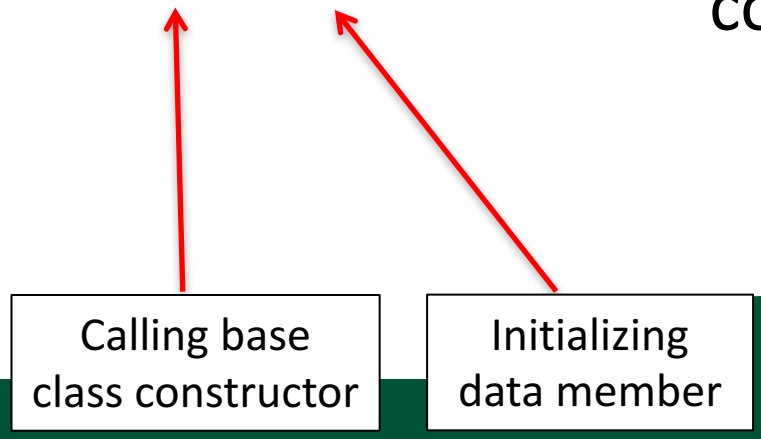
Initializers

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

class C : public A
{
  public:
    C(int x, int y) : A(y), z(x)  { };
  private:
    int z;
};

int main()
{
    C c(3,5);
}
```

- Needed because
  - Base class constructors are called before derived constructors & need to pass arguments in derived constructor to base class



Quiz

```
#include <stdio.h>

int doubler(int X)
{
    printf("In doubler\n");
    return 2*X;
}

class A
{
  public:
      A(int x) { printf("In A's constructor\n"); };
};

class B : public A
{
  public:
      B(int x) : A(doubler(x)) { printf("In B's constructor\n"); };
};

int main()
{
    B b(3);
}
```

```
fawcett:330 childs$ ./a.out
In doubler
In A's constructor
In B's constructor
```

What's the output?

The “is a” test

- Inhe… …s a” test
- Base class: Shape
- Derived types: Triangle, Rectangle, Circle
  - A triangle “is a” shape
  - A rectangle “is a” shape
  - A circle “is a” shape

I will do a live coding example of this next week, and will discuss how C++ implements virtual functions.

You can define an interface for Shapes, and the derived types can fill out that interface.

Multiple inheritance

- A class can inherit from more than one base type
- This happens when it “is a” for each of the base types
  - Inherits data members and methods of both base types

Multiple inheritance

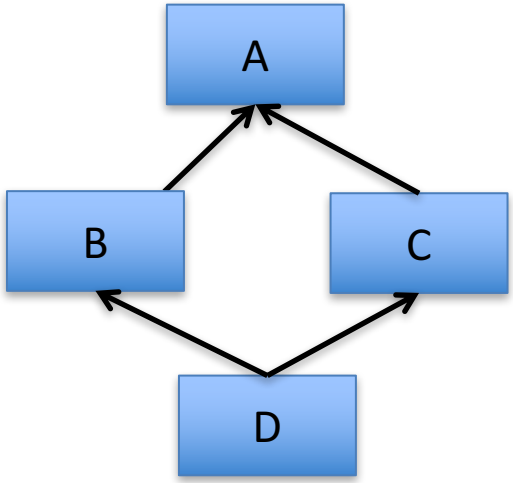
```
class Professor
{
    void Teach();
    void Grade();
    void Research();
};

class Father
{
    void Hug();
    void Discipline();
};

class Hank : public Father, public Professor
{
};
```

Diamond-Shaped Inheritance

- Base A, has derived types B and C, and D inherits from both B and C.
  - Which A is D dealing with??



- Diamond-shaped inheritance is controversial & really only for experts
  - (For what it is worth, we make heavy use of diamond-shaped inheritance in my project)

Diamond-Shaped Inheritance Example

```
class Person
{
    int X;
};

class Professor : public Person
{
    void Teach();
    void Grade();
    void Research();
};

class Father : public Person
{
    void Hug();
    void Discipline();
};

class Hank : public Father, public Professor
{
};
```

Diamond-Shaped Inheritance Pitfalls

```
#include <stdio.h>

class Person
{
  public:
     Person(int h) { hoursPerWeek = h; };
  protected:
    int hoursPerWeek;
};

class Professor : public Person
{
  public:
    Professor() : Person(90) { ; };
    void Teach();
    void Grade();
};

cla
{
  p
};
```

```
class Hank : public Father, public Professor
{
  public:
    int GetHoursPerWeek() { return hoursPerWeek; };
};

int main()
{
   Hank hrc;
   printf("HPW = %d\n", hrc.GetHoursPerWeek());
}
```

```
fawcett:330 childs$ g++ diamond_inheritance.C
diamond_inheritance.C: In member function ‘int Hank::GetHoursPerWeek()’:
diamond_inheritance.C:31: error: reference to ‘hoursPerWeek’ is ambiguous
diamond_inheritance.C:8: error: candidates are: int Person::hoursPerWeek
diamond_inheritance.C:8: error:                 int Person::hoursPerWeek
diamond_inheritance.C:31: error: reference to ‘hoursPerWeek’ is ambiguous
diamond_inheritance.C:8: error: candidates are: int Person::hoursPerWeek
diamond_inheritance.C:8: error:                 int Person::hoursPerWeek
```

Diamond-Shaped Inheritance Pitfalls

```
#include <stdio.h>

class Person
{
  public:
     Person(int h) { hoursPerWeek = h; };
  protected:
    int hoursPerWeek;
};

class Professor : public Person
{
  public:
    Professor() : Person(90) { ; };
    void Teach();
    void Grade();
    void Research();
};

class Father : public Person
{
  public:
    Father() : Person(20) { ; };
    void Hug();
    void Discipline();
};
```

```
class Hank : public Father, public Professor
{
  public:
    int GetHoursPerWeek() { return Professor::hoursPerWeek+
                                   Father::hoursPerWeek; };
};

int main()
{
   Hank hrc;
   printf("HPW = %d\n", hrc.GetHoursPerWeek());
}
```

```
fawcett:330 childs$ ./a.out
HPW = 110
```

This can get stickier with virtual functions.

You should avoid diamond-shaped inheritance until you feel really comfortable with OOP.

Pure Virtual Functions

- Pure Virtual Function: define a function to be part of the interface for a class, but do not provide a definition.
- Syntax: add “=0” after the function definition.
- This makes the class be “abstract”
  - It cannot be instantiated
- When derived types define the function, then are “concrete”
  - They can be instantiated

Pure Virtual Functions Example

```
class Shape
{
  public:
    virtual double GetArea(void) = 0;
};

class Rectangle : public Shape
{
  public:
    virtual double GetArea() { return 4; };
};

int main()
{
    Shape s;
    Rectangle r;
}
```

```
fawcett:330 childs$ g++ pure_virtual.C
pure_virtual.C: In function 'int main()':
pure_virtual.C:15: error: cannot declare variable 's' to be of abstract type 'Shape'
pure_virtual.C:2: note:   because the following virtual functions are pure within 'Shape':
pure_virtual.C:4: note:         virtual double Shape::GetArea()
```

Data Flow Networks

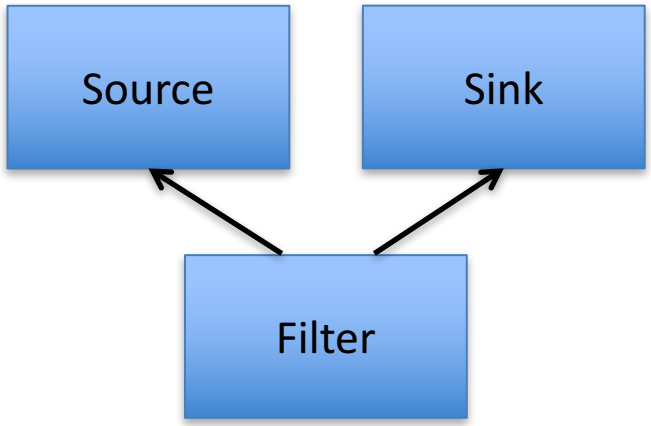
- This is not a C++ idea
- It is used for image processing, visualization, etc
- So we need to know it for Project C

Data Flow Overview

- Basic idea:
  - You have many modules
    - Hundreds!!
  - You compose modules together to perform some desired functionality
- Advantages:
  - Customizability
  - Design fosters interoperability between modules to the extent possible

Data Flow Overview

- Participants:
  - Source: a module that produces data
    - It creates an output
  - Sink: a module that consumes data
    - It operates on an input
  - Filter: a module that transforms input data to create output data

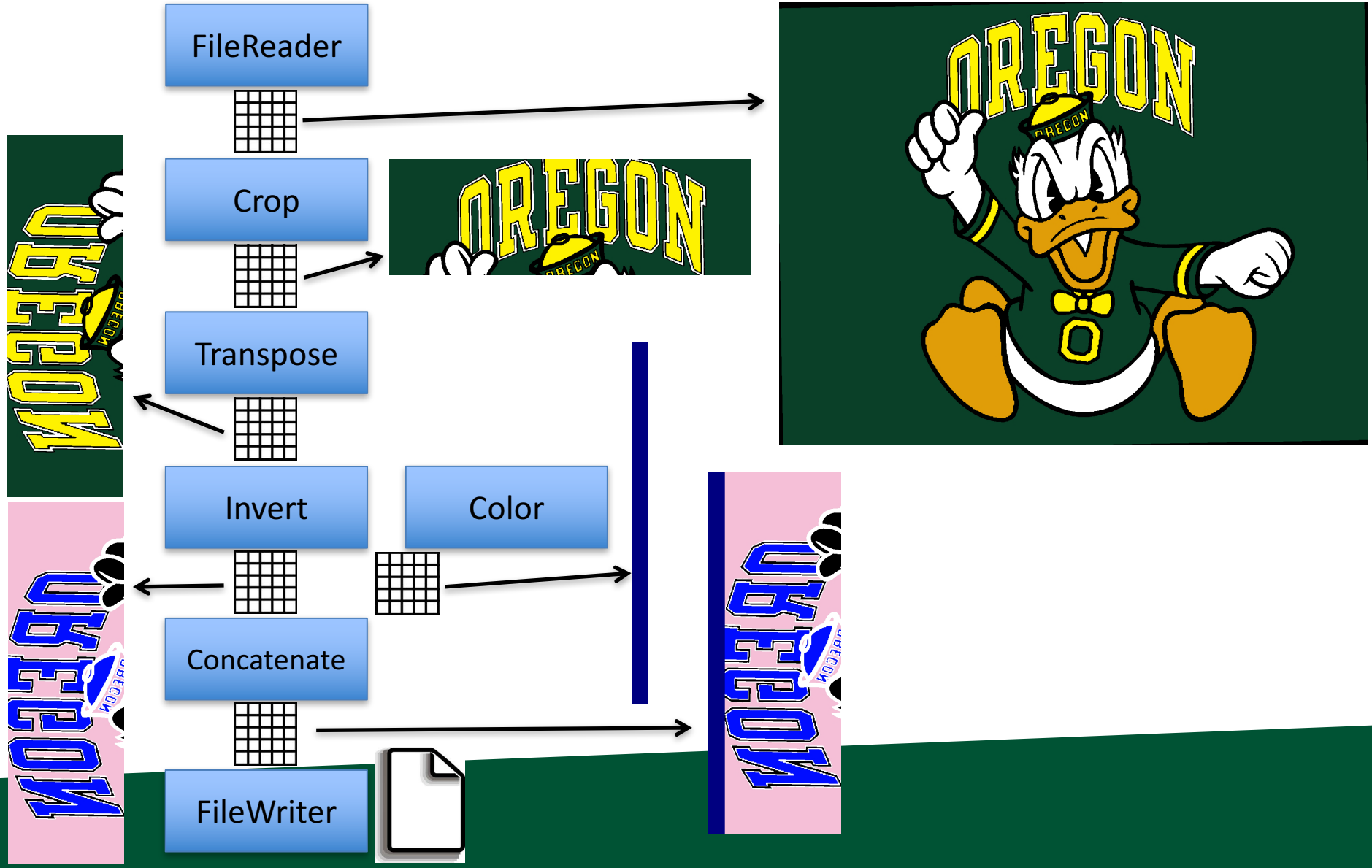


- Nominal inheritance hierarchy:
  - A filter “is a” source
  - A filter “is a” sink

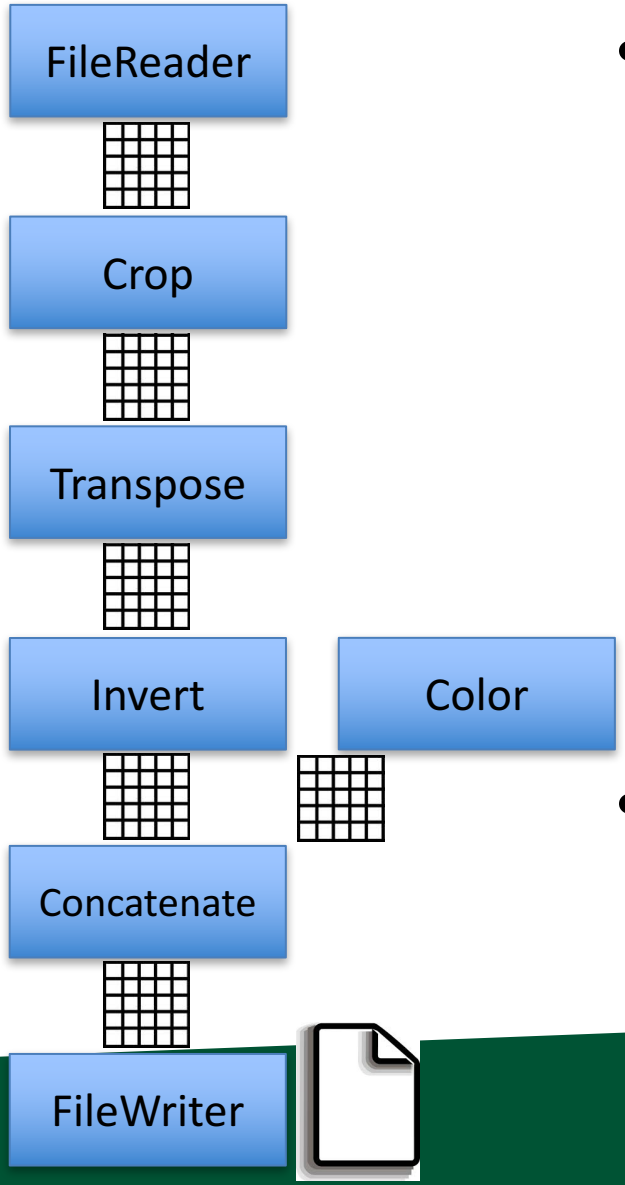
Example of data flow (image processing)

- Sources:
  - FileReader: reader from file
  - Color: generate image with one color
- Filters:
  - Crop: crop image, leaving only a sub-portion
  - Transpose: view image as a 2D matrix and transpose it
  - Invert: invert colors
  - Concatenate: paste two images together
- Sinks:
  - FileWriter: write to file

Example of data flow (image processing)



Example of data flow (image processing)



- Participants:
  - Source: a module that produces data
    - It creates an output
  - Sink: a module that consumes data
    - It operates on an input
  - Filter: a module that transforms input data to create output data
- Pipeline: a collection of sources, filters, and sinks connected together

Benefits of the Data Flow Design

- Extensib

What do you think the benefits are?

  - write infrastructure that knows about abstract types (source, sink, filter, and data object)
  - write as many derived types as you want
- Composable!
  - combine filters, sources and sinks in custom configurations

Drawbacks of Data Flow Design

What do you think the drawbacks are?

- Operations happen in stages
  - Extra memory needed for intermediate results
  - Not cache efficient
- Compartmentalization can limit possible optimizations

Data Flow Networks

- Idea:
  - Many modules that manipulate data
    - Called filters
  - Dynamically compose filters together to create "networks" that do useful things
  - Instances of networks are also called "pipelines"
    - Data flows through pipelines
  - There are multiple techniques to make a network "execute" … we won't worry about those yet

Data Flow Network: the players

- Source: produces data
- Sink: accepts data
  - Never modifies the data it accepts, since that data might be used elsewhere
- Filter: accepts data and produces data
  - A filter "is a" sink and it "is a" source

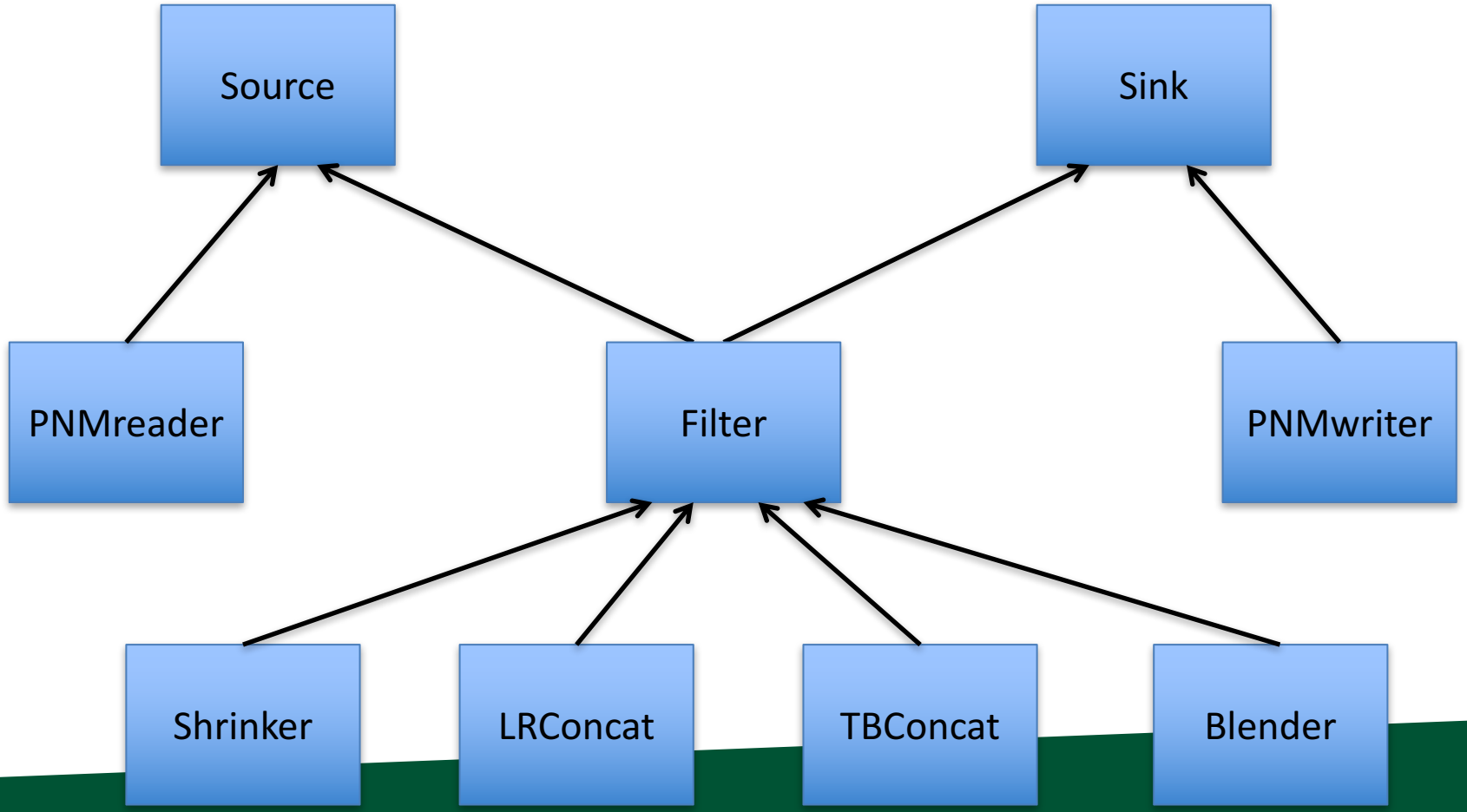
Source, Sink, and Filter are abstract types. The code associated with them facilitates the data flow.

There are concrete types derived from them, and they do the real work (and don't need to worry about data flow!).

Project 3C

- Due in one week
- 3D also due in one week
  - 3D not needed for 3E, 3F, etc.
  - So you can “skip”
  - But you will get 0 points for 3D if you “skip”

Assignment: make your code base be data flow networks with OOP



Topics for 3D

C++ lets you define operators

- You declare a method that uses an operator in conjunction with a class
  - +, -, /, !, ++, etc.
- You can then use your 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.

You can also do this with functions.

Example of operator overloading

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

    MyInt& operator++();

    int     Ge

  protected:
    int     my
};

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

Declare operator ++ will be overloaded for MyInt

We will learn more about operator overloading later in the quarter.

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

Call operator ++ on MyInt.

Define operator ++ for MyInt

New operators: << and >>

- "<<": Insertion operator
- ">>": Extraction operator
  - Operator overloading: you can define what it means to insert or extract your object.

- Often used in conjunction with "streams"
  - Recall our earlier experience with C streams
    - stderr, stdout, stdin
  - Streams are communication channels

cout: the C++ way of accessing stdout

```
fawcett:330 childs$ cat print.c
#include <stdio.h>

int main()
{
    printf("The answer is: ");
    printf("%d", 8);
    printf("\n");
}
fawcett:330 childs$ gcc print.c
fawcett:330 childs$ ./a.out
The answer is: 8
```

```
fawcett:330 childs$ cat printCPP.C
#include <iostream>

int main()
{
    std::cout << "The answer is: ";
    std::cout << 8;
    std::cout << "\n";
}
fawcett:330 childs$ g++ printCPP.C
fawcett:330 childs$ ./a.out
The answer is: 8
```

New header file (and no ".h"!)

New way of accessing stdout stream.

Insertion operation (<<)

cout is in the “standard” namespace

```
fawcett:330 childs$ cat printCPP.C
#include <iostream>

using std::cout;

int main()
{
    cout << "The answer is: ";
    cout << 8;
    cout << "\n";
}
fawcett:330 childs$ g++ printCPP.C
fawcett:330 childs$
```

“using” command puts the “cout” portion of the standard namespace (“std”) in the global namespace.

Don’t need “std::cout” any more…

endl: the C++ endline mechanism

- prints a newline
- flushes the stream
  - C version: fflush(stdout)
  - This is because printf doesn't always print when you ask it to.
    - It buffers the requests when you make them.
    - This is a problem for debugging!!

endl in action

```
fawcett:330 childs$ cat printCPP.C
#include <iostream>

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

int main()
{
    cout << "The answer is: ";
    cout << 8;
    cout << endl;
}
fawcett:330 childs$ g++ printCPP.C
fawcett:330 childs$
```

<< and >> have a return value

- ostream & ostream::operator<<(int);
  - (The signature for a function that prints an integer)
- The return value is itself
  - i.e., the cout object returns “cout”
- This allows you to combine many insertions (or extractions) in a single line.
  - This is called “cascading”.

Cascading in action

```
fawcett:330 childs$ cat printCPP.C
#include <iostream>

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

int main()
{
    cout << "The answer is: " << 8 << endl;
}
fawcett:330 childs$ g++ printCPP.C
fawcett:330 childs$
```

Putting it all together

```
fawcett:330 childs$ cat print.c
#include <stdio.h>

int main()
{
    printf("The answer is: ");
    printf("%d", 8);
    printf("\n");
}
fawcett:330 childs$ gcc print.c
fawcett:330 childs$ ./a.out
The answer is: 8
```

```
fawcett:330 childs$ cat printCPP.C
#include <iostream>

int main()
{
    std::cout << "The answer is: ";
    std::cout << 8;
    std::cout << "\n";
}
fawcett:330 childs$ g++ printCPP.C
fawcett:330 childs$ ./a.out
The answer is: 8
```

```
fawcett:330 childs$ cat print.C
#include <stdio.h>

int main()
{
    printf("The answer is: %d\n", 8);
}
fawcett:330 childs$ g++ print.C
fawcett:330 childs$
```

```
fawcett:330 childs$ cat printCPP.C
#include <iostream>

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

int main()
{
    cout << "The answer is: " << 8 << endl;
}
fawcett:330 childs$ g++ printCPP.C
fawcett:330 childs$
```

Three pre-defined streams

- cout <= => fprintf(stdout, …
- cerr <= => fprintf(stderr, …
- cin <= => fscanf(stdin, …

cin in action

```
fawcett:330 childs$ cat cin.C
#include <iostream>

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

int main()
{
    int X, Y, Z;
    cin >> X >> Y >> Z;
    cout << Z << ", " << Y << ", " << X << endl;
}
fawcett:330 childs$ ./a.out
3   5
4
4, 5, 3
```

cerr

- Works like cout, but prints to stderr
- Always flushes everything immediately!

"See the error"

```
fawcett:330 childs$ cat cerr.C
#include <iostream>

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

int main()
{
    int *X = NULL;
    stream << "The value is ";
    stream << *X << endl;
}
fawcett:330 childs$ g++ -Dstream=cerr cerr.C
fawcett:330 childs$ ./a.out
The value is Segmentation fault
fawcett:330 childs$ g++ -Dstream=cout cerr.C
fawcett:330 childs$ ./a.out
Segmentation fault
```

fstream

- ifstream: input stream that does file I/O
- ofstream: output stream that does file I/O

- Not lecturing on this, since it follows from:
  - C file I/O
  - C++ streams

http://www.tutorialspoint.com/cplusplus/cpp_files_streams.htm

Project 3D

- Assigned: today, 5/9
- Due: Weds, 5/16
- Important: if you skip this project, you will still be able to do future projects (3E, 3F, etc)
- Assignment:
  - Write PNMreaderCPP and PNMwriterCPP … new version of the file reader and writer that use fstream.

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

```
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

```
class MyDoublerClass
{
    int doubler(int X) { return 2*X; };
};
```

You should only do inlines within header files

```
fawcett:330 childs$ cat mydoubler.h
#ifndef MY_DOUBLER_H
#define MY_DOUBLER_H

class MyDoubler
{
  public:
    int Doubler(int X) { return 2*X; };
};


#endif
```

```
fawcett:330 childs$ cat mydoubler2.h
#ifndef MY_DOUBLER_H
#define MY_DOUBLER_H

class MyDoubler
{
  public:
};

int
MyDoubler::Doubler(int X)
{
   return 2*X;
}

#endif
```

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

Bonus Topics

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!!

Web pages

- ssh –l <user name> ix.cs.uoregon.edu
- cd public_html
- put something in index.html
- → it will show up as

  http://ix.cs.uoregon.edu/~<username>

Web pages

- You can also exchange files this way
  - scp file.pdf <username>@ix.cs.uoregon.edu:~/public_html
  - point people to http://ix.cs.uoregon.edu/~<username>/file.pdf

Note that ~/public_html/dir1 shows up as http://ix.cs.uoregon.edu/~<username>/dir1

("~/dir1" is not accessible via web)