CS100
Introduction to Programming

Lectures 20: Advanced topics II

Outline

- Preprocessor instructions
- Exceptions
- Design patterns
- Debugging

Outline

- Preprocessor instructions
- Exceptions
- Design patterns
- Debugging

Preprocessor directives

- Preprocessor directives are lines in the code preceded by a hash-tag (#)
- They are processed by the ”pre-processor”
  - Resolved before actual compilation starts
  - They will be replaced by something else
  - They define code to be replaced by something else
- Example: include-guards!

```
#ifndef MYCLASS_HPP_
#define MYCLASS_HPP_
//…
#endif
```

Defines

- `#define`
- Can be used to define constants
  - Example:

```c
#define PI 3.1415

//…

int main() {
//…
       float circum = 2.0f * radius * PI;
//…
}
```

Defines

- #define
- Does not need a value
  - Can simply mark a variable as defined

```
#define DEBUG_MODE

//…

int main() {
#ifdef DEBUG_MODE
       if( !(index < vec.size()) )
             cout << “Access out of bounds\n”;
#endif
}
```

Defines

- `#define`
- Can be set through the console
  - Example: `g++ -DDEBUG_MODE problem1.cpp -o main`

```
int main() {
#ifdef DEBUG_MODE
       if( !(index < vec.size()) )
             cout << “Access out of bounds\n”;
#endif
}
```

Defines

- `#define` vs `static const`
  - `#define` consumes no memory
  - `#define` has no type, so can be assigned flexibly
  - `static const` can be scoped
  - `static const` has a single, clearly defined type, which may also be an advantage
  - `static const` enables pointers

Macros

- `#define` can be used to define functions
- Syntax:

  ```
  #define fctName(param1,param2) ([fct using params])
  ```

- Example:

  ```
  #define getrandom(min, max) ((rand()%(int)(((max) + 1)-(min)))+ (min))
  ```

typedef

- Sometimes we may have long type-names
- Example:

```
std::vector< std::vector< MyMathLibrary::Types::Matrix<double> > >
```

- We can introduce an alias for this type:

```
typdef std::vector< std::vector< MyMathLibrary::Types::Matrix<double> > >;
```

- Can be in global, class, or function scope

Outline

- Preprocessor instructions
- Exceptions
- Design patterns
- Debugging

What is an exception?

- An exception or exceptional event is an event that occurs during the execution of a program that disrupts the normal flow of instructions
- The following will cause exceptions:
  - Accessing an out-of-bounds array element
  - Writing into a read-only file
  - Trying to read beyond the end of a file
  - Sending illegal arguments to a method
  - Performing illegal arithmetic (e.g divide by 0)
  - Hardware failures
  - …

Handling exceptions

- Basic idea:
    - Check for exceptional events
    - Deal with them
- Example of simple exception handling

```
T & operator()( int r, int c ) {
    if( !(r < rows()) || !(c < rows()) ) {
          std::cout << “Error: attempt to access “
          std::cout << “element out of bounds\n”;
          return m_data[0][0];
    }
    // normal code
    return m_data[r][c];
}
```

«Exceptions»

- You can use a **try-catch** block to handle exceptions that are thrown

```
try {
    // code that might throw exception
}
catch ([Type of Exception] e) {
// what to do if exception is thrown
}
```

«Exceptions»

- You can use multiple catch blocks to catch exceptions

```
try {
    // code that might throw exception
}
catch ([Type of Exception 1] e) {
// what to do if exception 1 is thrown
}
catch ([Type of Exception 2] e) {
// what to do if exception 2 is thrown
}
catch ([Type of Exception 3] e) {
// what to do if exception 3 is thrown
}
```

«Exceptions»

- Example

```
template<class T>
class Matrix {
//…
T & operator()( int r, int c ) {
    try {
          if( !(r < rows()) || !(c < rows()) )
                throw “Access out of bounds”;
          // normal code
          return m_data[r][c];
    }
    catch(const char * msg) {
          std::cout << msg << “\n”;
          return m_dummyVal;
    }
}
//…
```

Throwing Exceptions

- Use the **throw** statement to throw an exception

  -
    ```
    if(ptr.equals(null))
     throw “Null ptr exception”;
    ```
- The throw statement requires a single argument: a throwable object
- **Throwable** objects are

  primitive types

  instances of any subclass of `std::exception`

«Exceptions»

- Can use more complicated types
- Need to overload std::exception

```
class OutOfBoundsExc : public std::exception {
public:
    OutOfBoundsExc( int r, int c, int rows, int cols ) :
          m_r(r), m_c(c), m_rows(rows), m_cols(cols) {}
    void print() {
          std::cout << “Attempt to access element (”
                    << r << “,” << c << “) in a matrix ”
                    << “of size ” << m_rows
                    << “x” << m_cols;
    }
private:
    int m_r;
    int m_c;
    int m_rows;
    int m_cols;
}
```

«Exceptions»

- Using the new type

```
T & operator()( int r, int c ) {
    try {
          if( !(r < rows()) || !(c < rows()) )
                 throw OutOfBoundsExc(r, c, rows(), cols());
          // normal code
          return m_data[r][c];
    }
    catch(OutOfBoundsExc & e) {
          e.print();
          return m_dummyVal;
    }
}
```

«Exceptions»

- Exceptions can be caught anywhere!

```
template<class T>
class Matrix {
//…
  T & operator()( int r, int c )  {
    if( !(r < rows()) || !(c < rows()) )
      throw OutOfBoundsExc(r,c,rows(),cols());
    // normal code
    return m_data[r][c];
  }
//…
}

int main() {
    Matrix<double> mat(5,5);
    double val;
    try { val = mat(3,5); }
    catch( OutOfBoundsExc & e ) { e.print(); }
    return 0;
}
```

Dummy val no longer needed!

Why handling exceptions

- Compilation cannot find all errors
- To separate error detection, reporting, and handling
  - Reporting/handling are separated from regular code
  - Increases code clarity
  - We defer how to worry about errors to somewhere else
- Group and differentiate error types
  - Write error handlers that handle very specific exceptions

Outline

- Preprocessor instructions
- Exceptions
- Design patterns
- Debugging

What is a design pattern?

- A solution for a recurring problem in a large OOP system

- "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice"
  - Charles Alexander

Types of design patterns

- Design patterns can be grouped into 3 categories:

- **Creational Patterns** *(abstracting the object-instantiation process)*
  - **Factory Method** Abstract Factory Singleton
  - Builder Prototype

- **Structural Patterns** *(how objects/classes can be combined)*
  - **Adapter** Bridge Composite
  - Decorator Facade Flyweight
  - Proxy

- **Behavioral Patterns** *(communication between objects)*
  - Command Interpreter Iterator
  - Mediator Observer State
  - Strategy Chain of Responsibility Visitor
  - Template Method

Factory Method

- Define an interface for creating an object, but let subclasses decide which class to instantiate
- Factory Method lets a class defer instantiation to subclasses
- Defines a virtual constructor (i.e. the factory)
- operator **`new`** is considered harmful

Factory Method

- Example

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

class Circle: public Shape {
public:
  void print() {cout << ”I am a Triangle\n";}
};

class Square: public Shape {
public:
  void print() {cout << ”I am a Square\n";}
};

class Rectangle: public Shape {
public:
  void print() {cout << ”I am a Rectangle\n";}
};
```

Factory Method

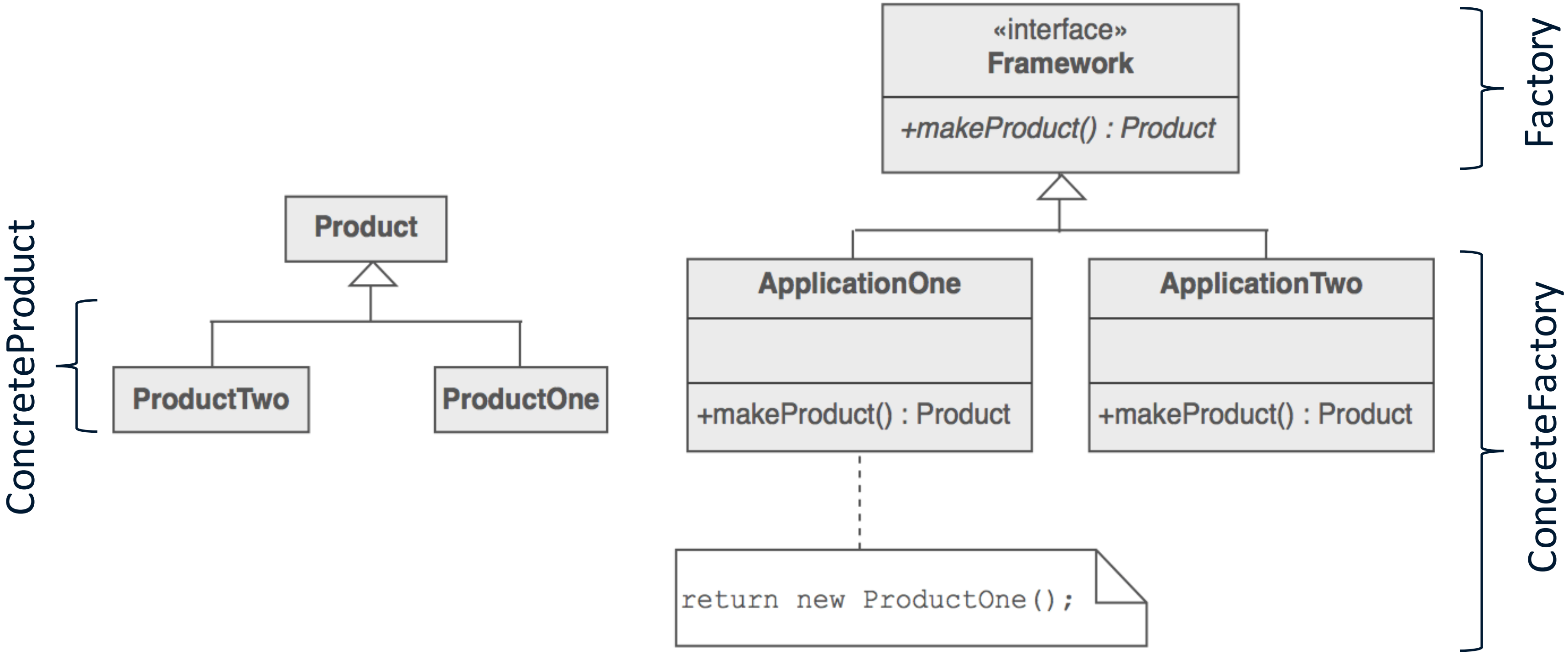
- Example (cont.)

```
int main() {
  vector<Shape*> shapes;
  int choice;
  while(true) {
    cout << "Circle(1) Square(2) Rectangle(3) ";
    cout << "Done(0): ";
    cin >> choice;
    if(choice == 0)
      break;
    else if (choice == 1)
      shapes.push_back(new Circle());
    else if (choice == 2)
      shapes.push_back(new Square());
    else
      shapes.push_back(new Rectangle());
  }

  for(int i = 0; i < shapes.size(); i++)
    shapes[i]->print();
  for(int i = 0; i < shapes.size(); i++)
    delete shapes[i];
}
```

Factory Method

- Analysis of the example
  - Programmer made effort to decouple the user of Shape from concrete derived classes by applying polymorphism
  - However:
    - There remains coupling
    - The creation of Shapes requires the user to call a concrete constructor of a derived class (e.g. **`shapes.push_back(new Circle());`**)
  - Factory method:
    - Define a virtual constructor (i.e. the factory) in the Shape base-class or even a separate Factory class
    - Let the user use Shapes without ever having to get in touch with concrete derived classes

Factory Method



«interface»
Framework

+makeProduct() : Product

ApplicationOne

+makeProduct() : Product

ApplicationTwo

+makeProduct() : Product

```
return new ProductOne();
```

Product

ProductTwo

ProductOne

ConcreteProduct

Factory

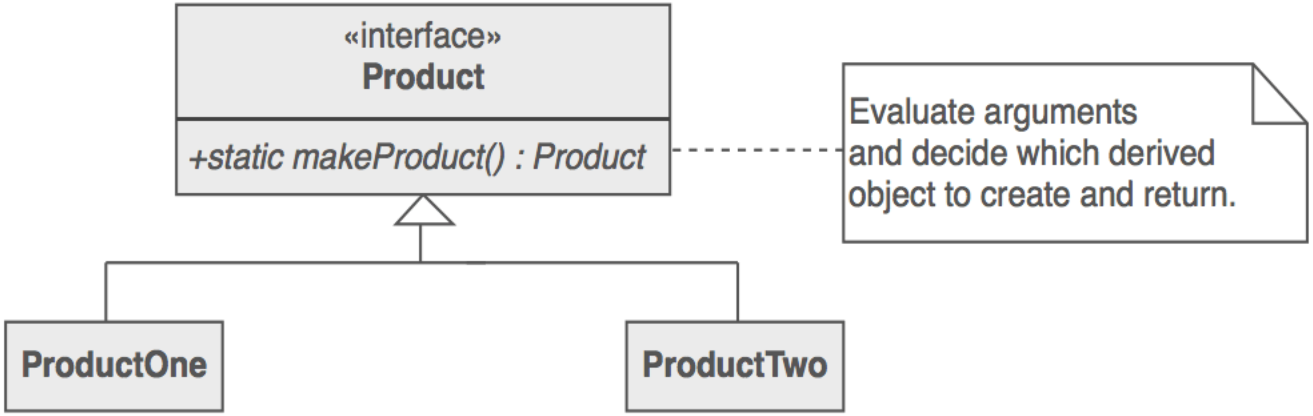
ConcreteFactory

Factory Method

- Vocabulary
  - Product
    - Defines the interface of objects the factory method creates
  - ConcreteProduct
    - Implements the product interface
  - Factory
    - Declares the factory method which returns an object of type Product
    - May contain a default implementation of the factory method
    - Creator relies on its subclasses to define the factory method so that it returns an instance of the appropriate Concrete Product.
  - ConcreteFactory
    - Overrides factory method to return instance of ConcreteProduct

Factory Method

- Simplified implementation
  - Static function in the base class of the Product
  - Implements the virtual constructor



Evaluate arguments and decide which derived object to create and return.

Factory Method

- Back to example

```
class Shape {
public:
  static Shape * make_shape(int choice);
  virtual void print() = 0;
};

class Circle: public Shape { //… }
class Square: public Shape { //… }
class Rectangle: public Shape { //… }


Shape *
Shape::make_shape(int choice) {
  if(choice == 1)
    return new Circle();
  else if(choice == 2)
    return new Square();
  else
    return new Rectangle();
}
```

Factory Method

- Back to example (cont.)

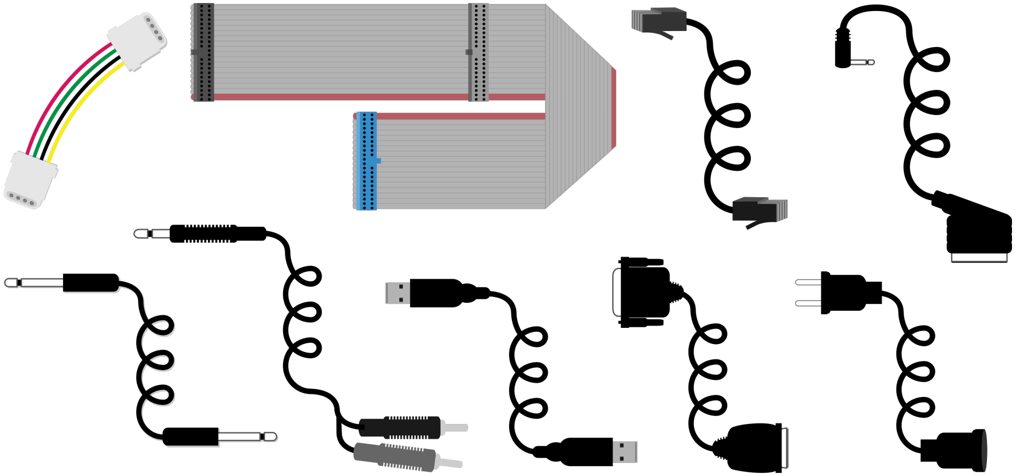
```
int main() {
  vector<Shape*> shapes;
  int choice;
  while(true) {
    cout << "Circle(1) Square(2) Rectangle(3) ";
    cout << "Done(0): ";
    cin >> choice;
    if(choice == 0)
      break;
    shapes.push_back(Shape::make_shape(choice));
  }

  for(int i = 0; i < shapes.size(); i++)
    shapes[i]->print();
  for(int i = 0; i < shapes.size(); i++)
    delete shapes[i];
}
```

Factory Method

- Use the Factory Method when
  - a class can’t anticipate the class of objects it must create
  - a class wants its subclasses to specify the objects it creates
  - classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate

Adapter Pattern

- Problem: We have an object that contains the functionality we need, but not in the way we want to use it.
  - Cumbersome / unpleasant to use. Prone to bugs.
- Solution: An Adapter!
  - An object that fits another object into a given interface



Adapter Pattern

- Example: Suppose we have a legacy Rectangle class that we would want to use

```
class LegacyRectangle {
public:
  LegacyRectangle( float x1, float y1, float x2, float y2) {
    m_x1 = x1; m_y1 = y1; m_x2 = x2; m_y2 = y2;
  }
  float area() {
    return (m_x2-m_x1)*(m_y2-m_y1);
  }
private:
  float m_x1;
  float m_x2;
  float m_y1;
  float m_y2;
};
```

Adapter Pattern

- The instance that wants to use LegacyRectangle however only has the position and size of the rectangle, and also expects everything to be communicated in inch

```
//desired interface
class Rectangle {
public:
  virtual float areaSqInch(); //unit: square-inch
};
```

Adapter Pattern

- Making and using the adapter

```
// Adapter wrapper
class RectangleAdapter: public Rectangle, private LegacyRectangle {
public:
  RectangleAdapter(float x, float y, float w, float h) :
      LegacyRectangle(
          x/39.37f,y/39.37f,(x + w)/39.37f,(y + h)/39.37f) {}
  virtual float areaSqInch() {
    float areaSqM = area();
    return areaSqM * 1550.0f;
  }
};

int main() {
  Rectangle *r = new RectangleAdapter(120.0f, 200.0f, 6.0f, 4.0f);
  cout << r->areaSqInch() << “\n”;
}
```

Outline

- Preprocessor instructions
- Exceptions
- Design patterns
- Errors and Debugging

Understanding Errors

```
hw2.c:87:7: error: ‘foo’ undeclared
```

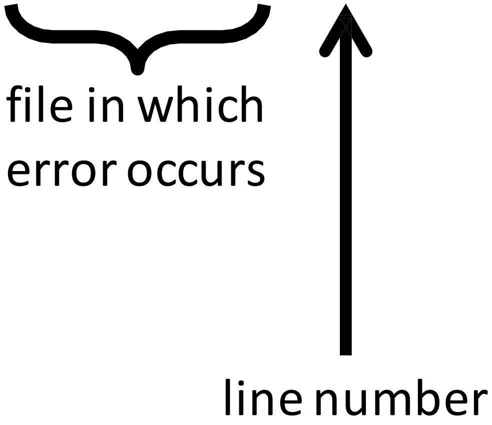
Understanding Errors

```
hw2.c:87:7: error: ‘foo’ undeclared
```

file in which
error occurs

Understanding Errors

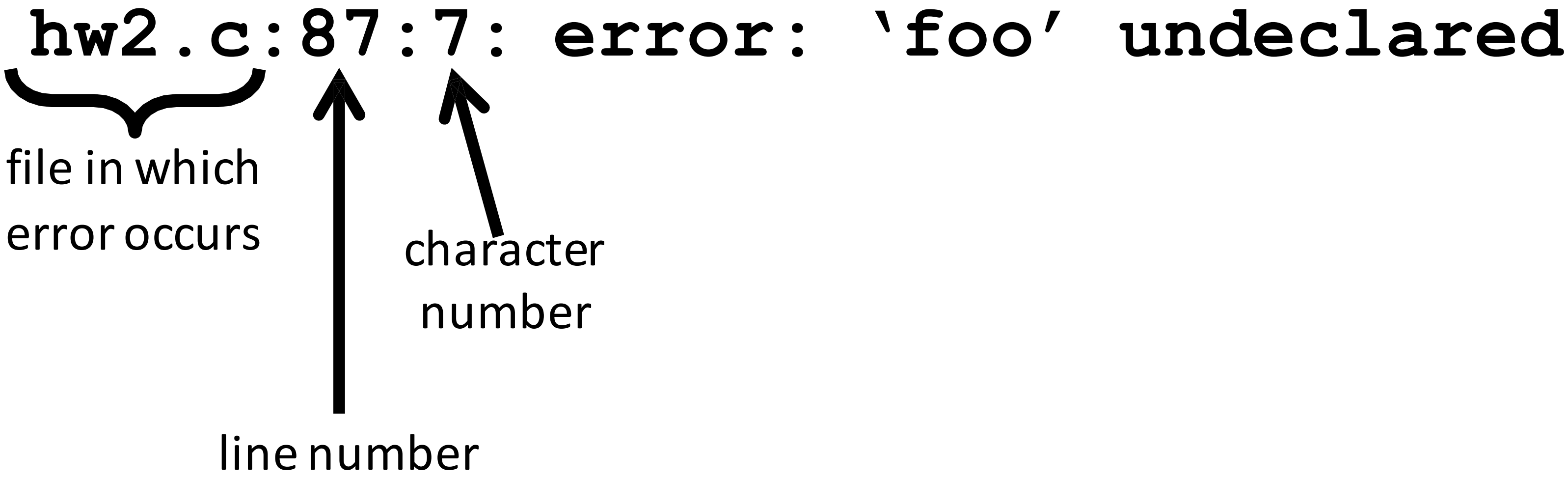
```
hw2.c:87:7: error: ‘foo’ undeclared
```



file in which error occurs

line number

Understanding Errors



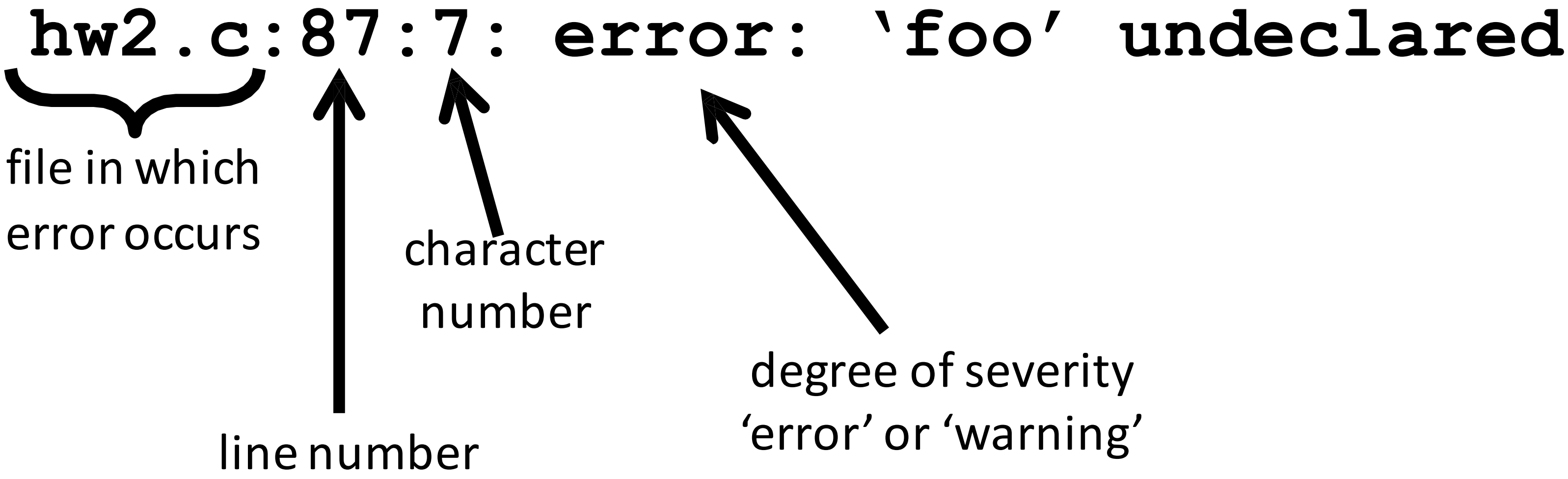
```
hw2.c:87:7: error: ‘foo’ undeclared
```

file in which error occurs

character number

line number

Understanding Errors



```
hw2.c:87:7: error: ‘foo’ undeclared
```

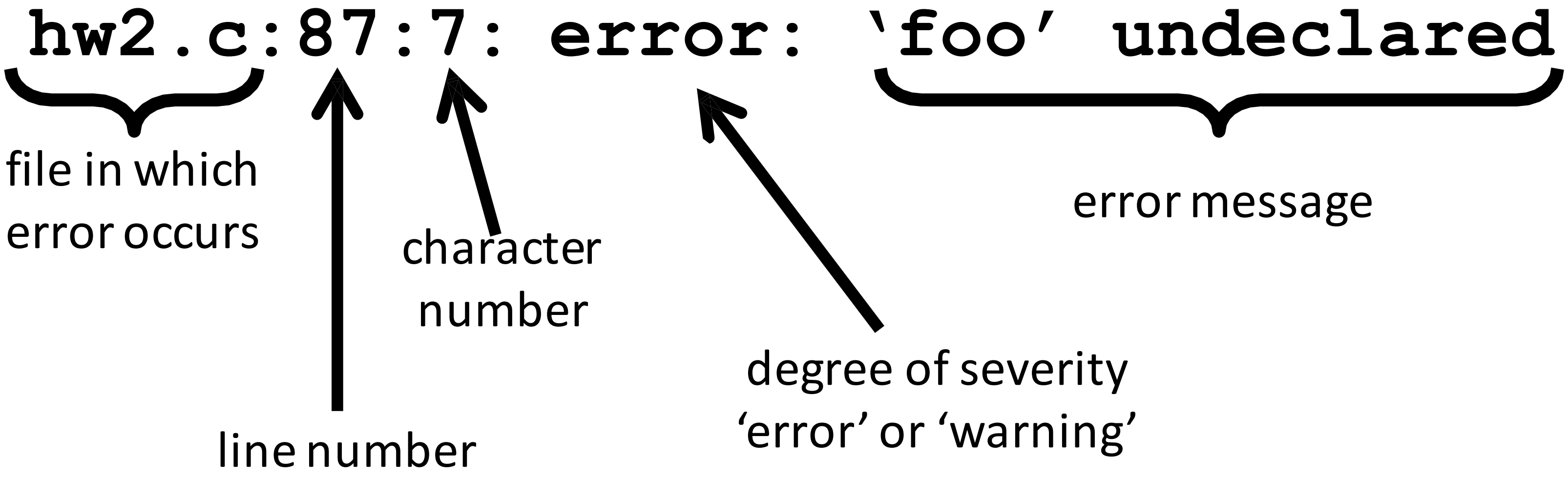
file in which error occurs

line number

character number

degree of severity ‘error’ or ‘warning’

Understanding Errors



```
hw2.c:87:7: error: ‘foo’ undeclared
```

file in which error occurs

line number

character number

degree of severity ‘error’ or ‘warning’

error message

#1 Rule of Debugging

- start with the **very first** error or warning

- recompile every time an error is fixed
  - errors will cascade
  - and de-cascade when fixed!

Cascading Errors

```
int numStudnts;
for (i = 0; i < numStudents; i++) {
  total += grades[i];
}
avg = total/numStudents;
```

Cascading Errors

```
int numStudnts;
for (i = 0; i < numStudents; i++) {
   total += grades[i];
}
avg = total/numStudents;
```

```
> gcc -Wall average.c
```

Cascading Errors

```
int numStudnts;
for (i = 0; i < numStudents; i++) {
   total += grades[i];
}
avg = total/numStudents;
```

```
> gcc -Wall average.c
```

- the **`-Wall`** flag shows all of warnings

Cascading Errors

```
int numStudnts;
for (i = 0; i < numStudents; i++) {
  total += grades[i];
}
avg = total/numStudents;
```

```
> gcc -Wall average.c
average.c:5:5: warning: unused variable ‘numStudnts’
average.c:22:17: error: ‘numStudents’ undeclared
average.c:25:13: error: ‘numStudents’ undeclared
```

Cascading Errors

```
int numStudnts;
for (i = 0; i < numStudents; i++) {
  total += grades[i];
}
avg = total/numStudents;
```

```
> gcc -Wall average.c
average.c:5:5: warning: unused variable ‘numStudnts’
[redacted]
```

Cascading Errors

```
int numStudnts;
for (i = 0; i < numStudents; i++) {
   total += grades[i];
}
avg = total/numStudents;
```

```
> gcc -Wall average.c
average.c:5:5: warning: unused variable ‘numStudnts’
average.c:22:17: error: ‘numStudents’ undeclared
average.c:25:13: error: ‘numStudents’ undeclared
```

Cascading Errors

```
int numStudents;
for (i = 0; i < numStudents; i++) {
  total += grades[i];
}
avg = total/numStudents;
```

Cascading Errors

```
int numStudents;
for (i = 0; i < numStudents; i++) {
  total += grades[i];
}
avg = total/numStudents;
```

```
> gcc -Wall average.c
```

Cascading Errors

```
int numStudents;
for (i = 0; i < numStudents; i++) {
  total += grades[i];
}
avg = total/numStudents;
```

```
> gcc -Wall average.c
```

- got rid of all 3 errors!

When Errors Occur

- compile time
  - pretty easy (normally typos or simple mistakes)
- linking
  - slightly harder (could be easy, could require rethinking how your code is laid out)
- run time
  - UGH (often difficult to pinpoint, and sometimes hard to spot at all)
  - best bet is to use a debugger

Common Compiler Errors

```
hw2.c:87:7: error: ‘foo’ undeclared
```

- if `foo` is a **variable**:
  - forgot to declare
  - misspelled (on declaration or on use)
- if `foo` is a **function**:
  - forgot to `#include` file containing the prototype
  - misspelled (on declaration or on use)

Common Compiler Errors

```
hw2.c:37:6: warning: unused variable
    ‘bar’
```

- variable was declared but not used
  - normally because variable declaration has a typo
  - if you’re in the midst of writing code, this warning may be *temporarily* acceptable
    - haven’t had a chance to use the variable yet

Common Compiler Errors

```
hw2.c:54: warning: suggest
    parentheses around assignment
    used as truth value
```

- often a mistake inside a control statement
  - you meant to use **==** not **=**
  - (you want equivalency, not assignment)

Common Compiler Errors

```
hw2.c: 51: error: expected ‘;’
   before ‘for’
```

- missing semicolon on previous line of code
- ‘for’ is simply the word directly following the missing semicolon
  - could be ‘int’ or ‘if’ or a variable name, etc

Common Linker Errors

```
hw4.o: In function ‘main’:
hw4.c:91: undefined reference to ‘Fxn’
```

- linker can’t find code for ‘Fxn’ in any .o file
  - forgot to link `.o` file
  - misspelled named of Fxn
  - parameter list is different
    - differences between prototype/definition/call

Common Linker Errors

```
/usr/lib64/gcc/[...]/crt1.o: In function
    ‘_start’:
/home/[...]/start.S:119: undefined
    reference to main
```

- you compiled a file that does not contain a **`main()`**
- without using the **`-c`** flag to indicate separate compilation

Error messages can be very long …

```
> gcc -Wall structs.c
In file included from /usr/include/stdio.h:33:0,
                 from structs.c:6:
/usr/lib64/gcc/x86_64-suse-linux/4.7/include/stddef.h:213:1: error:
expected '=', ',', ';', 'asm' or '__attribute__' before 'typedef'
In file included from /usr/include/stdio.h:74:0,
                 from structs.c:6:
/usr/include/libio.h:307:3: error: unknown type name 'size_t'
/usr/include/libio.h:311:67: error: 'size_t' undeclared here (not in a
function)
/usr/include/libio.h:339:62: error: expected declaration specifiers or
'...' before 'size_t'
/usr/include/libio.h:348:6: error: expected declaration specifiers or '...'
before 'size_t'
/usr/include/libio.h:470:19: error: expected '=', ',', ';', 'asm' or
'__attribute__' before '_IO_sgetn'
In file included from structs.c:6:0:
/usr/include/stdio.h:319:35: error: expected declaration specifiers or
'...' before 'size_t'
/usr/include/stdio.h:325:47: error: expected declaration specifiers or
'...' before 'size_t'
/usr/include/stdio.h:337:20: error: expected declaration specifiers or
'...' before 'size_t'
/usr/include/stdio.h:344:10: error: expected declaration specifiers or
'...' before 'size_t'
/usr/include/stdio.h:386:44: error: expected declaration specifiers or
'...' before 'size_t'
/usr/include/stdio.h:390:45: error: expected declaration specifiers or
'...' before 'size_t'
/usr/include/stdio.h:666:11: error: expected declaration specifiers or
'...' before 'size_t'
/usr/include/stdio.h:669:9: error: expected declaration specifiers or
'...' before 'size_t'
/usr/include/stdio.h:679:8: error: expected declaration specifiers or
'...' before 'size_t'
/usr/include/stdio.h:709:15: error: expected '=', ',', ';', 'asm' or
'__attribute__' before 'fread'
/usr/include/stdio.h:715:15: error: expected '=', ',', ';', 'asm' or
'__attribute__' before 'fwrite'
/usr/include/stdio.h:737:15: error: expected '=', ',', ';', 'asm' or
'__attribute__' before 'fread_unlocked'
/usr/include/stdio.h:739:15: error: expected '=', ',', ';', 'asm' or
'__attribute__' before 'fwrite_unlocked'
In file included from structs.c:9:0:
/usr/include/string.h:43:8: error: expected declaration specifiers or '...'
before 'size_t'
/usr/include/string.h:46:56: error: expected declaration specifiers or
'...' before 'size_t'
/usr/include/string.h:55:18: error: expected declaration specifiers or
'...' before 'size_t'
/usr/include/string.h:62:42: error: expected declaration specifiers or
'...' before 'size_t'
/usr/include/string.h:65:56: error: expected declaration specifiers or
'...' before 'size_t'
/usr/include/string.h:92:48: error: expected declaration specifiers or
'...' before 'size_t'
/usr/include/string.h:129:39: error: expected declaration specifiers or
'...' before 'size_t'
/usr/include/string.h:137:9: error: expected declaration specifiers or
'...' before 'size_t'
/usr/include/string.h:143:57: error: expected declaration specifiers or
'...' before 'size_t'
/usr/include/string.h:150:15: error: expected '=', ',', ';', 'asm' or
'__attribute__' before 'strxfrm'
In file included from structs.c:9:0:
/usr/include/string.h:165:15: error: expected '=', ',', ';', 'asm' or
'__attribute__' before 'strxfrm_l'
/usr/include/string.h:180:45: error: expected declaration specifiers or
'...' before 'size_t'
/usr/include/string.h:281:15: error: expected '=', ',', ';', 'asm' or
'__attribute__' before 'strcspn'
/usr/include/string.h:285:15: error: expected '=', ',', ';', 'asm' or
'__attribute__' before 'strspn'
/usr/include/string.h:395:15: error: expected '=', ',', ';', 'asm' or
'__attribute__' before 'strlen'
/usr/include/string.h:402:15: error: expected '=', ',', ';', 'asm' or
'__attribute__' before 'strnlen'
/usr/include/string.h:423:12: error: expected declaration specifiers or
'...' before 'size_t'
/usr/include/string.h:447:33: error: expected declaration specifiers or
'...' before 'size_t'
/usr/include/string.h:451:53: error: expected declaration specifiers or
'...' before 'size_t'
/usr/include/string.h:455:31: error: expected declaration specifiers or
'...' before 'size_t'
/usr/include/string.h:458:54: error: expected declaration specifiers or
'...' before 'size_t'
/usr/include/string.h:536:61: error: expected declaration specifiers or
'...' before 'size_t'
/usr/include/string.h:573:34: error: expected declaration specifiers or
'...' before 'size_t'
/usr/include/string.h:576:39: error: expected declaration specifiers or
'...' before 'size_t'
In file included from structs.c:11:0:
/usr/include/stdlib.h:139:15: error: expected '=', ',', ';', 'asm' or
'__attribute__' before '__ctype_get_mb_cur_max'
In file included from structs.c:11:0:
/usr/include/stdlib.h:331:4: error: expected declaration specifiers or
'...' before 'size_t'
/usr/include/stdlib.h:361:4: error: expected declaration specifiers or
'...' before 'size_t'
/usr/include/stdlib.h:465:22: error: expected declaration specifiers or
'...' before 'size_t'
/usr/include/stdlib.h:467:22: error: expected declaration specifiers or
'...' before 'size_t'
/usr/include/stdlib.h:467:38: error: expected declaration specifiers or
'...' before 'size_t'
/usr/include/stdlib.h:479:36: error: expected declaration specifiers or
'...' before 'size_t'
In file included from /usr/include/stdlib.h:491:0,
                 from structs.c:11:
/usr/include/alloca.h:32:22: error: expected declaration specifiers or
'...' before 'size_t'
In file included from structs.c:11:0:
/usr/include/stdlib.h:497:22: error: expected declaration specifiers or
'...' before 'size_t'
/usr/include/stdlib.h:502:45: error: expected declaration specifiers or
'...' before 'size_t'
/usr/include/stdlib.h:502:65: error: expected declaration specifiers or
'...' before 'size_t'
/usr/include/stdlib.h:755:9: error: expected declaration specifiers or
'...' before 'size_t'
/usr/include/stdlib.h:755:25: error: expected declaration specifiers or
'...' before 'size_t'
/usr/include/stdlib.h:760:34: error: expected declaration specifiers or
'...' before 'size_t'
/usr/include/stdlib.h:760:50: error: expected declaration specifiers or
'...' before 'size_t'
/usr/include/stdlib.h:839:6: error: expected declaration specifiers or
'...' before 'size_t'
/usr/include/stdlib.h:842:6: error: expected declaration specifiers or
'...' before 'size_t'
/usr/include/stdlib.h:846:31: error: expected declaration specifiers or
'...' before 'size_t'
/usr/include/stdlib.h:850:31: error: expected declaration specifiers or
'...' before 'size_t'
/usr/include/stdlib.h:859:36: error: expected declaration specifiers or
'...' before 'size_t'
/usr/include/stdlib.h:863:34: error: expected declaration specifiers or
'...' before 'size_t'
/usr/include/stdlib.h:870:15: error: expected '=', ',', ';', 'asm' or
'__attribute__' before 'mbstowcs'
/usr/include/stdlib.h:873:15: error: expected '=', ',', ';', 'asm' or
'__attribute__' before 'wcstombs'
```

… but not too hard to fix

- Follow the message til the original calling point
  - …
  - In file included from …
  - …
  - In file included from …
  - …
  - Instantiated here …
  - …
  - Instantiated here …
  - Error message

Debugging Basics

- if the error’s not clear from just looking at the code, you can try:

- inserting probe statements with printf
  - (but adding a printf might change your error!)
- rubber duck debugging
- googling the error message
- using a debugger

Debuggers

- see what is going on “inside” the program
  - more powerful and accurate than printf() probes
- examine individual variables (value & address)
  - can change variable’s value on the fly
- step through code line by line
  - can skip blocks of code you don’t want to see

Using GDB

- must use the ‘**-g**’ flag when compiling

- open program for testing using command line:

  ```
  gdb hw2
  ```

- GDB – Gnu Project Debugger (text based)

Using GDB

- debugger allows you to:
  - add breakpoints to stop the program at specific points
  - use ‘print’ or ‘display’ to show values (or addresses) of variables
  - step through code line by line