Programming Languages
Part 2

Copyright © 2020 by
Robert M. Dondero, Ph.D.
Princeton University
Objectives

• You will learn/review:
  – Subsets of Java and Python...
  – That are appropriate for COS 333...
  – Through example programs
    • Example 1 in Java, Python
    • Example 2 in Java, Python
    • …
Agenda

• What are the data types and operators?
• How do I do terminal I/O?
• What are the statements?
• How do I express a program as multiple files?
“Circle” Programs

• The job:
  – Read a circle’s radius from stdin
  – Write the circle’s diameter and circumference to stdout
“Circle” in Java

• See `Circle.java`
  – Terminal input via `Scanner` class
  – Data types: `int`, `double`
  – Assignment statement
  – Operators: `=`, `*`, `+`, `cast`
  – `printf()` method
  – Catching exceptions via `try...catch`

• Generalizing
  – Data types, operators, and terminal I/O...
## Some Java Primitive Data Types

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Size</th>
<th>Example Literals</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>1 bit</td>
<td>false, true</td>
</tr>
<tr>
<td>byte</td>
<td>1 byte</td>
<td>(byte)-128, (byte)127</td>
</tr>
<tr>
<td>short</td>
<td>2 bytes</td>
<td>(short)-32768, (short)32767</td>
</tr>
<tr>
<td>int</td>
<td>4 bytes</td>
<td>-2147483648, 2147483647</td>
</tr>
<tr>
<td>long</td>
<td>8 bytes</td>
<td>-9223372036854775808L, 9223372036854775807L</td>
</tr>
<tr>
<td>float</td>
<td>4 bytes</td>
<td>123.456F</td>
</tr>
<tr>
<td>double</td>
<td>8 bytes</td>
<td>123.456</td>
</tr>
</tbody>
</table>
## Java Operators

<table>
<thead>
<tr>
<th>Operators</th>
<th>(Precedence) Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>method(params)</td>
<td>(1) Method call</td>
</tr>
<tr>
<td>a[i]</td>
<td>(1) Array element selector</td>
</tr>
<tr>
<td>obj.field, obj.method()</td>
<td>(1) Object member selector</td>
</tr>
<tr>
<td>class.field, class.method()</td>
<td>(1) Class member selector</td>
</tr>
<tr>
<td>i++, ++i, i--, --i</td>
<td>(2) Increment, decrement</td>
</tr>
<tr>
<td>+x, −x</td>
<td>(2) Unary plus, unary minus</td>
</tr>
<tr>
<td>~i, !b</td>
<td>(2) Bitwise NOT, logical NOT</td>
</tr>
<tr>
<td><strong>new</strong> Class()</td>
<td>(3) Object creation</td>
</tr>
<tr>
<td>(type)</td>
<td>(3) Typecast</td>
</tr>
<tr>
<td>x*y, x/y, i%j</td>
<td>(4) Multiplication, division, remainder</td>
</tr>
</tbody>
</table>
# Java Operators

<table>
<thead>
<tr>
<th>Operators</th>
<th>(Precedence) Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x+y, x-y</code></td>
<td>(5) Addition, subtraction</td>
</tr>
<tr>
<td><code>i&lt;&lt;j, i&gt;&gt;j, i&gt;&gt;&gt;j</code></td>
<td>(6) Left shift, right shift, right shift zero fill</td>
</tr>
<tr>
<td><code>x&lt;y, x&gt;y, x&lt;=y, x&gt;=y</code></td>
<td>(7) Relational</td>
</tr>
<tr>
<td><code>obj instanceof Class</code></td>
<td>(7) instanceof</td>
</tr>
<tr>
<td><code>x==y, x!=y</code></td>
<td>(8) relational</td>
</tr>
<tr>
<td><code>i&amp;j</code></td>
<td>(9) Bitwise AND</td>
</tr>
<tr>
<td><code>i^j</code></td>
<td>(10) Bitwise exclusive OR</td>
</tr>
<tr>
<td>`i</td>
<td>j`</td>
</tr>
<tr>
<td><code>b1&amp;&amp;b2</code></td>
<td>(12) Logical AND</td>
</tr>
<tr>
<td>`b1</td>
<td></td>
</tr>
</tbody>
</table>
## Java Operators

<table>
<thead>
<tr>
<th>Operators</th>
<th>(Precedence) Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>b?x:y</code></td>
<td>(14) Conditional expression</td>
</tr>
<tr>
<td><code>x=y</code></td>
<td>(15) Assignment</td>
</tr>
<tr>
<td><code>x+=y, x-=y, x*=y, x/=y, x%=y</code></td>
<td>(15) Assignment</td>
</tr>
<tr>
<td>`i&amp;=j, i^=j, i</td>
<td>=j`</td>
</tr>
<tr>
<td><code>i&lt;=j, i&gt;=j, i&gt;&gt;=j</code></td>
<td>(15) Assignment</td>
</tr>
</tbody>
</table>
Java Terminal I/O

• Reading from stdin:

```java
import java.util.Scanner;
...
Scanner scanner = new Scanner(System.in);
int i = scanner.nextInt();
double d = scanner.nextDouble();
String s = scanner.nextLine();
```
Java Terminal I/O

• Writing to stdout:

```java
System.out.print(i);
System.out.println(d);
System.out.println(s);
System.out.printf("%d %f %s", i, d, s);
```

• Writing to stderr:

```java
System.err.print(i);
System.err.println(d);
System.err.println(s);
System.err.printf("%d %f %s", i, d, s);
```
“Circle” in Python

• See `circle.py`
  – Terminal input via `input()`
  – String literals: '...' or "...
  – No variable declarations
  – Data types: `int`, `float`
  – Type conversions via `int()` and `float()`
  – Assignment statement
  – Operators: * (multiplication), % (string formatting)
  – `print()` is variadic
  – Catching exceptions via `try...except`
“Circle” in Python

• Generalizing
  – Conversion functions: `int(obj)`, `float(obj)`, `bool(obj)`
  – String formatting: conversion specifications are as in C
  – Data types, operators, and terminal I/O...
Some Python Built-In Data Types

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Example Literals</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>1, 23, 3493, 01, 027, 06645, 0x1, 0x17, 0xDA5</td>
</tr>
<tr>
<td></td>
<td>(no theoretical size limit)</td>
</tr>
<tr>
<td>float</td>
<td>0., 0.0, .0, 1.0, 1e0, 1.0e0</td>
</tr>
<tr>
<td></td>
<td>(corresponds to C double)</td>
</tr>
<tr>
<td>bool</td>
<td>False, True</td>
</tr>
</tbody>
</table>
## Python Operators

<table>
<thead>
<tr>
<th>Operators</th>
<th>(Priority) Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{key:expr, ...}</code></td>
<td>(1) <strong>Dictionary</strong> creation</td>
</tr>
<tr>
<td><code>[expr,...]</code></td>
<td>(2) <strong>List</strong> creation</td>
</tr>
<tr>
<td><code>(expr,...)</code></td>
<td>(3) <strong>Tuple</strong> creation, or just parentheses</td>
</tr>
<tr>
<td><code>f(expr,...)</code></td>
<td>(4) <strong>Function call</strong></td>
</tr>
<tr>
<td><code>x[startindex:stopindex]</code></td>
<td>(5) <strong>Slicing</strong> (for sequences)</td>
</tr>
<tr>
<td><code>x[index]</code></td>
<td>(6) <strong>Indexing</strong> (for containers)</td>
</tr>
<tr>
<td><code>x.attr</code></td>
<td>(7) <strong>Attribute reference</strong></td>
</tr>
<tr>
<td><code>x**y</code></td>
<td>(8) <strong>Exponentiation</strong></td>
</tr>
<tr>
<td><code>~x</code></td>
<td>(9) <strong>Bitwise NOT</strong></td>
</tr>
<tr>
<td><code>+x, -x</code></td>
<td>(10) <strong>Unary plus</strong>, <strong>unary minus</strong></td>
</tr>
<tr>
<td><code>x*y, x/y, x//y, x%y</code></td>
<td>(11) <strong>Mult</strong>, <strong>div</strong>, <strong>truncating div</strong>, <strong>remainder</strong></td>
</tr>
<tr>
<td></td>
<td><em>(or string formatting)</em></td>
</tr>
<tr>
<td><code>x+y, x-y</code></td>
<td>(12) <strong>Addition</strong>, <strong>subtraction</strong></td>
</tr>
</tbody>
</table>
Aside: Python Division

Beware of division operators:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 // 2</td>
<td>2</td>
</tr>
<tr>
<td>5 / 2</td>
<td>2.5</td>
</tr>
<tr>
<td>float(5) / 2</td>
<td>2.5</td>
</tr>
<tr>
<td>4 / 2</td>
<td>2.0</td>
</tr>
<tr>
<td>float(4) / 2</td>
<td>2.0</td>
</tr>
</tbody>
</table>

Suggestion: use only boldfaced forms

And that’s quite different from Python 2!!!
## Python Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>(Priority) Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{x&lt;&lt;i, x&gt;&gt;i}</td>
<td>(13) Left-shift, right-shift</td>
</tr>
<tr>
<td>\texttt{x&amp;y}</td>
<td>(14) Bitwise AND</td>
</tr>
<tr>
<td>\texttt{x^y}</td>
<td>(15) Bitwise XOR</td>
</tr>
<tr>
<td>\texttt{x</td>
<td>y}</td>
</tr>
<tr>
<td>\texttt{x&lt;y, x&lt;=y, x&gt;y, x&gt;=y}</td>
<td>(17) Relational</td>
</tr>
<tr>
<td>\texttt{x==y, x!=y}</td>
<td>(17) Relational</td>
</tr>
<tr>
<td>\texttt{x is y, x is not y}</td>
<td>(18) \textbf{Identity} tests</td>
</tr>
<tr>
<td>\texttt{x in y, x not in y}</td>
<td>(19) \textbf{Membership} tests</td>
</tr>
<tr>
<td>\texttt{not x}</td>
<td>(20) Logical NOT</td>
</tr>
<tr>
<td>\texttt{x and y}</td>
<td>(21) Logical AND</td>
</tr>
<tr>
<td>\texttt{x or y}</td>
<td>(22) Logical OR</td>
</tr>
</tbody>
</table>
Python Terminal I/O

- Reading from stdin:

```python
str = input()
str = input(promptStr)

from sys import stdin
...
str = stdin.readline()
```
Python Terminal I/O

• Writing to stdout:

```python
print(str)
print(str, end=' ')
print(str1, str2, str3)
print(str1, str2, str3, end='')
```

• Writing to stderr:

```python
from sys import stderr
...
print(str, file=stderr)
print(str, end=' ', file=stderr)
print(str1, str2, str3, end=' ', file=stderr)
```
Agenda

• What are the data types and operators?
• How do I do terminal I/O?
• **What are the statements?**
• How do I express a program as multiple files?
“Euclid” Programs

• The job
  – Read two integers from stdin
  – Write their gcd and lcm to stdout
“Euclid” in Java

• See Euclid.java
  – Control flow statements: if, while, return
  – Throwing exceptions via throw

• Generalizing
  – Statements...
Java Statements

Expression statement
  \textit{expr};

Declaration statement
  \textit{modifiers datatype var [= expr]}
  [, \textit{var [= expr]}]...;
  \textbf{Common modifiers: const, static}

Compound statement
  \{ \textit{statement; statement; ...} \}
if statement
  if (booleanExpr) statement else statement;

switch statement
  switch (intExpr)
  { case (intvalue1):
    statement; ...; break;
    case (intvalue2):
    statement; ...; break;
    ...
    ... break;
    default:
    statement; ...;
  }
Java Statements

while statement
    while (booleanExpr) statement;

do statement
    do statement while (booleanExpr);

for statement
    for (initExpr; booleanExpr; incrExpr)
        statement;

“Foreach” statement
    for (type var : IterableOrArrayObject)
        statement;
Java Statements

```java
break statement
    break;

continue statement
    continue;

return statement
    return;
    return expr;
```
Java Statements

try...catch statement
  try
  {
    statement;
    statement;
    ...
  }
  catch (ExceptionType e)
  {
    statement;
    statement;
    ...
  }

throw statement
  throw object;
“Euclid” in Python

• See `euclid.py`
  – *Unpacking assignment* statement
  – Control flow statements: `if, while, return`
  – Throwing exceptions via `raise`

• Generalizing
  – Statements...
### Python Statements

**Assignment statements**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>var = expr</code></td>
<td>Simple assignment</td>
</tr>
<tr>
<td><code>var += expr</code></td>
<td>Add and assign</td>
</tr>
<tr>
<td><code>var -= expr</code></td>
<td>Subtract and assign</td>
</tr>
<tr>
<td><code>var *= expr</code></td>
<td>Multiply and assign</td>
</tr>
<tr>
<td><code>var /= expr</code></td>
<td>Divide and assign</td>
</tr>
<tr>
<td><code>var //= expr</code></td>
<td>Floor divide and assign</td>
</tr>
<tr>
<td><code>var %= expr</code></td>
<td>Modulus and assign</td>
</tr>
<tr>
<td><code>var **= expr</code></td>
<td>Power and assign</td>
</tr>
<tr>
<td><code>var &amp;= expr</code></td>
<td>Bitwise AND and assign</td>
</tr>
<tr>
<td>`var</td>
<td>= expr`</td>
</tr>
<tr>
<td><code>var ^= expr</code></td>
<td>Bitwise XOR and assign</td>
</tr>
<tr>
<td><code>var &gt;&gt;= expr</code></td>
<td>Right shift and assign</td>
</tr>
<tr>
<td><code>var &lt;&lt;= expr</code></td>
<td>Left shift and assign</td>
</tr>
</tbody>
</table>
Unpacking assignment statement

```
var1, var2, ... = iterable
```

No-op statement

```
pass
```

assert statement

```
assert expr, message
```
Python Statements

Function call statement

\[ f(expr, name=expr, ...) \]

return statement

return

return expr
Python Statements

```
if statement
    if expr:
        statement(s)
    elif expr:
        statement(s)
    else:
        statement(s)
```

False, 0, None, '', '', [], (), and {} indicate logical FALSE
Any other value indicates logical TRUE
Python Statements

while statement
  while expr:
    statement(s)

False, 0, None, '', '', [], (), and {} indicate logical FALSE
Any other value indicates logical TRUE
for statements
    for var in range(startint, stopint):
        statements
    for var in range(stopint):
        statements
    for var in iterable:
        statements
    for key, value in mapping.items():
        statements

break statement
    break
break

continue statement
    continue
try...except statement
    try:
        statement(s)
    except [ExceptionType [as var]]:
        statement(s)

raise statement
    raise ExceptionType(str)
Agenda

• What are the data types and operators?
• How do I do terminal I/O?
• What are the statements?
• How do I express a program as multiple files?
“IntMath” Programs

• The job
  – Same as “Euclid” programs, except…
  – Factor gcd() and lcm() defs into distinct “IntMath” module
“IntMath” in Java

• See IntMath.java, TestIntMath.java
  – TestIntMath.java is a client of the IntMath module
“IntMath” in Java

• Building and running
  – $ javac TestIntMath.java
    • Automatically builds IntMath.java
    • Directory containing IntMath.java must be in CLASSPATH env var
    • "." is in CLASSPATH by default
  – $ java TestIntMath
“IntMath” in Python

• See intmath1.py, testintmath1.py
  – testintmath1.py is a client of the intmath1 module
  – testintmath1.py must import from intmath1.py
“IntMath” in Python

• Building and running

  – $ python testintmath1.py
    • Automatically compiles/interprets intmath1.py
    • Directory containing intmath1.py must be in PYTHONPATH env var
    • "." is in PYTHONPATH by default
“IntMath” in Python

• See intmath2.py, testintmath2.py
  – Parameter type validation

  – Which is better, intmath1.py or intmath2.py?
Aside: Python Duck Typing

• Observation:
  – Python doesn’t care about the types of a function’s parameters
  – Python cares only that a function’s parameters respond to the messages that the function sends to them
Aside: Python Duck Typing

• Observation:
  – In other words…
  – Python uses *duck typing*

“When I see a bird that walks like a duck and swims like a duck and quacks like a duck I call that bird a duck.”

-- J. W. Riley
Aside: Python Duck Typing

• **Style 1:** Don’t validate parameter types
  – Rationale: Validating parameter types prohibits fortuitous use of functions/methods on parameters of unanticipated types
  – Rationale: Validating parameter types consumes run-time
  – **So intmath1.py is better**
Aside: Python Duck Typing

• **Style 2**: Validate parameter types
  – Rationale: When a parameter is of an unanticipated type, function/method may yield:
    • Errors at low call levels rather than immediately
    • No errors and incorrect results!!!
  – In our example:
    • intmath2.py safeguards against str or float parameters
    • intmath1.py does not
  – **So intmath2.py is better**
Aside: Python Duck Typing

• Controversy!
  – Generally style 1 (do not validate parameter types) is considered more “Pythonic”
  – … So that’s what we’ll use in COS 333
Aside: Python Duck Typing

• Commentary
  – Application-dependent code
    • Maybe need not validate parameter types
  – Application-independent code
    • Maybe should validate parameter types
    • The negative of accidental incorrect results outweighs the positive of fortuitous use of duck typing!!!
Aside: Python Duck Typing

• Commentary
  – **Small** projects:
    • Maybe need not validate parameter types
  – **Large** projects:
    • Maybe should validate parameter types
Aside: Python Duck Typing

• Commentary
  – But if you feel the need to validate parameter types, should you use Python anyway???
Summary

• We have covered the answers to these questions for Java and Python:
  – What are the data types and operators?
  – How do I do terminal I/O?
  – What are the statements?
  – How do I express a program as multiple files?
Appendix:
C Programming
“Circle” in C

• See `circle.c`
  – Data types: `int`, `double`
  – Operators: `=`, `*`, `+`, `cast`
  – Terminal input via `scanf()`
  – Lack of exception handling
    • Must check function return value instead

• Generalizing
  – Data types, operators, and terminal I/O...
# C Primitive Data Types

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Size</th>
<th>Example Literals</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1 byte</td>
<td>(char)-128, (char)127, 'a', '\n', '\t', '\0'</td>
</tr>
<tr>
<td>unsigned char</td>
<td>1 byte</td>
<td>(unsigned char) 0, (unsigned char)256</td>
</tr>
<tr>
<td>short</td>
<td>2 bytes?</td>
<td>(short)-32768, (short)32767</td>
</tr>
<tr>
<td>unsigned short</td>
<td>2 bytes?</td>
<td>(short)0, (short)65535</td>
</tr>
<tr>
<td>int</td>
<td>4 bytes?</td>
<td>-2147483648, 2147483647</td>
</tr>
<tr>
<td>unsigned int</td>
<td>4 bytes?</td>
<td>0U, 4294967295U, 0123, 0x123</td>
</tr>
<tr>
<td>long</td>
<td>4 bytes?</td>
<td>-2147483648L, 2147483647L</td>
</tr>
<tr>
<td>unsigned long</td>
<td>4 bytes?</td>
<td>0UL, 4294967295UL, 0123L, 0x123L</td>
</tr>
<tr>
<td>float</td>
<td>4 bytes?</td>
<td>123.456F</td>
</tr>
<tr>
<td>double</td>
<td>8 bytes?</td>
<td>123.456</td>
</tr>
<tr>
<td>long double</td>
<td>12 bytes?</td>
<td>123.456L</td>
</tr>
</tbody>
</table>
## C Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>(Precedence) Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>fun(params)</td>
<td>(1) Function call</td>
</tr>
<tr>
<td>a[i]</td>
<td>(1) Array element selector</td>
</tr>
<tr>
<td>s.f</td>
<td>(1) Structure field selector</td>
</tr>
<tr>
<td>ps-&gt;f</td>
<td>(1) Structure dereference and field selector</td>
</tr>
<tr>
<td>(type)</td>
<td>(2) Typecast</td>
</tr>
<tr>
<td>*p, &amp;x</td>
<td>(2) Dereference, address of</td>
</tr>
<tr>
<td>i++, ++i, i--, --i</td>
<td>(2) Increment, decrement</td>
</tr>
<tr>
<td>+x, -x</td>
<td>(2) Unary positive, unary negative</td>
</tr>
<tr>
<td>~i</td>
<td>(2) Bitwise NOT</td>
</tr>
<tr>
<td>!i</td>
<td>(2) Logical NOT</td>
</tr>
<tr>
<td>sizeof(type)</td>
<td>(2) sizeof</td>
</tr>
</tbody>
</table>
# C Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>(Precedence) Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>x*y, x/y, i%j</td>
<td>(3) Multiplication, division, remainder</td>
</tr>
<tr>
<td>x+y, x-y</td>
<td>(4) Addition, subtraction</td>
</tr>
<tr>
<td>i&lt;&lt;j, i&gt;&gt;j</td>
<td>(5) Left-shift, right-shift</td>
</tr>
<tr>
<td>x, y, x&gt;y, x&lt;=y, x&gt;=y</td>
<td>(6) Relational</td>
</tr>
<tr>
<td>x==y, x!=y</td>
<td>(7) Relational</td>
</tr>
<tr>
<td>i&amp;j</td>
<td>(8) Bitwise AND</td>
</tr>
<tr>
<td>i^j</td>
<td>(9) Bitwise EXCLUSIVE OR</td>
</tr>
<tr>
<td>i</td>
<td>j</td>
</tr>
<tr>
<td>i&amp;&amp;j</td>
<td>(11) Logical AND</td>
</tr>
<tr>
<td>i</td>
<td></td>
</tr>
<tr>
<td>i?x:y</td>
<td>(13) Conditional expression</td>
</tr>
</tbody>
</table>
## C Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>(Precedence) Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=y</td>
<td>(14) Assignment</td>
</tr>
<tr>
<td>x+=y, x-=y, x*=y, x/=y</td>
<td>(14) Assignment</td>
</tr>
<tr>
<td>i&amp;=j, i^=j, i</td>
<td>=j</td>
</tr>
<tr>
<td>i&lt;&lt;=j, i&gt;&gt;=j</td>
<td>(14) Assignment</td>
</tr>
<tr>
<td>x,y</td>
<td>(15) Sequence</td>
</tr>
</tbody>
</table>
C Terminal I/O

• Reading from stdin:
  - `iValuesRead = scanf("%d", &i);`
  - `iValuesRead = scanf("%lf", &d);`
  - `s = fgets(s, size, stdin)`
C Terminal I/O

- **Writing to stdout:**
  - `printf("%d", i);`
  - `printf("%f", d);`
  - `fputs(s, stdout);`

- **Writing to stderr:**
  - `fprintf(stderr, "%d", i);`
  - `fprintf(stderr, "%f", d);`
  - `fputs(s, stderr);`
“Euclid” in C

• See euclid.c
  – Control flow statements: if, while, return

• Generalizing
  – Statements...
C Statements

Expression statement

```c
expr;
```

Declaration statement

```c
modifiers datatype var [= expr]
[, var [= expr]]...;
Common modifiers: const, static
```

Compound statement

```c
{ statement; statement; ...}
```
C Statements

If statement

```c
if (intOrPtrExpr) statement [else statement]
```

0 or NULL pointer => false; anything else => true

Switch statement

```c
switch (intexpr)
{
    case (intvalue1):
        statement; ...; break;
    case (intvalue2):
        statement; ...; break;
    ...
    default:
        statement; ...
}
```
C Statements

While statement
   while (intOrPtrExpr) statement;

Do...while statement
   do statement while (intOrPtrExpr);

For statement
   for (initexpr; intOrPtrExpr; increxpr) statement;
C Statements

Break statement
break;

Continue statement
continue;

Return statement:
return;
return expr;
“IntMath” in C

• See *intmath.h, intmath.c, testintmath.c*
  – *intmath.h* is IntMath module *interface*
    • Protection against multiple inclusion
    • Function names begin with module name
  – *intmath.c* is IntMath module *implementation*
    • `#include intmath.h` so compiler can enforce consistency
  – *testintmath.c* is IntMath module *client*
    • `#include intmath.h` so compiler can check calls of IntMath_gcd() and IntMath_lcm()
“IntMath” in C

• Building:
  − $ gcc intmath.c testintmath.c -o testintmath

• Running:
  − $ testintmath