And You Thought There Couldn't be More C++

C++

Outline

- Multi-File Programs
- makefiles

Multi-File Programs

- Advantages
  - If you write classes in separate files (like in Java) you can use that class in multiple programs
    - Increases reusability
  - If you want to change something in a class, change it in one place and all programs using it reflect change
  - If multiple people are working on a project, makes sense to have separate files
    - Real world – many programmers will be working on one project

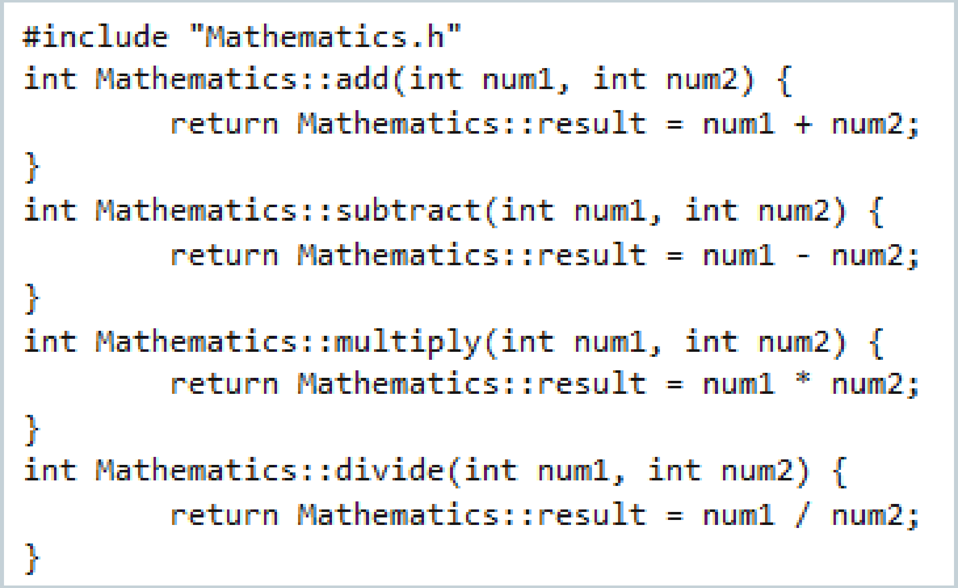
Multi-File Programs

- Class Libraries
  - We have included header files
    - These refer to files in the C++ library
  - Advantages
    - Reuse of code written by (and tested by) other people
  - You can write your own libraries if you like

Multi-File Programs

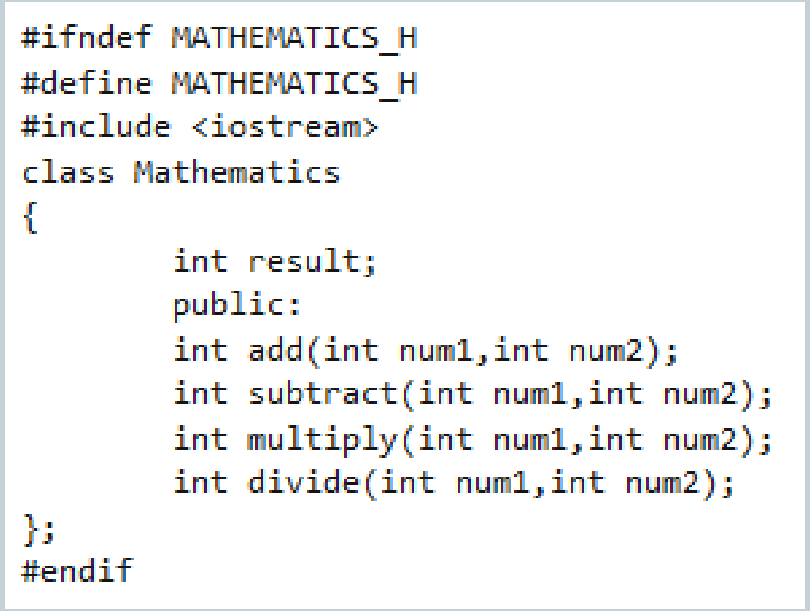
- Header files
  - Contain information about classes and functions
  - Usually .h extension – for “header”
  - Can then use the preprocessor directive #include to use them
    - e.g. #include <stdio.h>
    - stdio is a library of i/o files, stdio.h is the header file that describes all the available functions
  - Essentially describes the API to the functions and classes

Multi-File Programs



```
#include "Mathematics.h"
int Mathematics::add(int num1, int num2) {
        return Mathematics::result = num1 + num2;
}
int Mathematics::subtract(int num1, int num2) {
        return Mathematics::result = num1 - num2;
}
int Mathematics::multiply(int num1, int num2) {
        return Mathematics::result = num1 * num2;
}
int Mathematics::divide(int num1, int num2) {
        return Mathematics::result = num1 / num2;
}
```

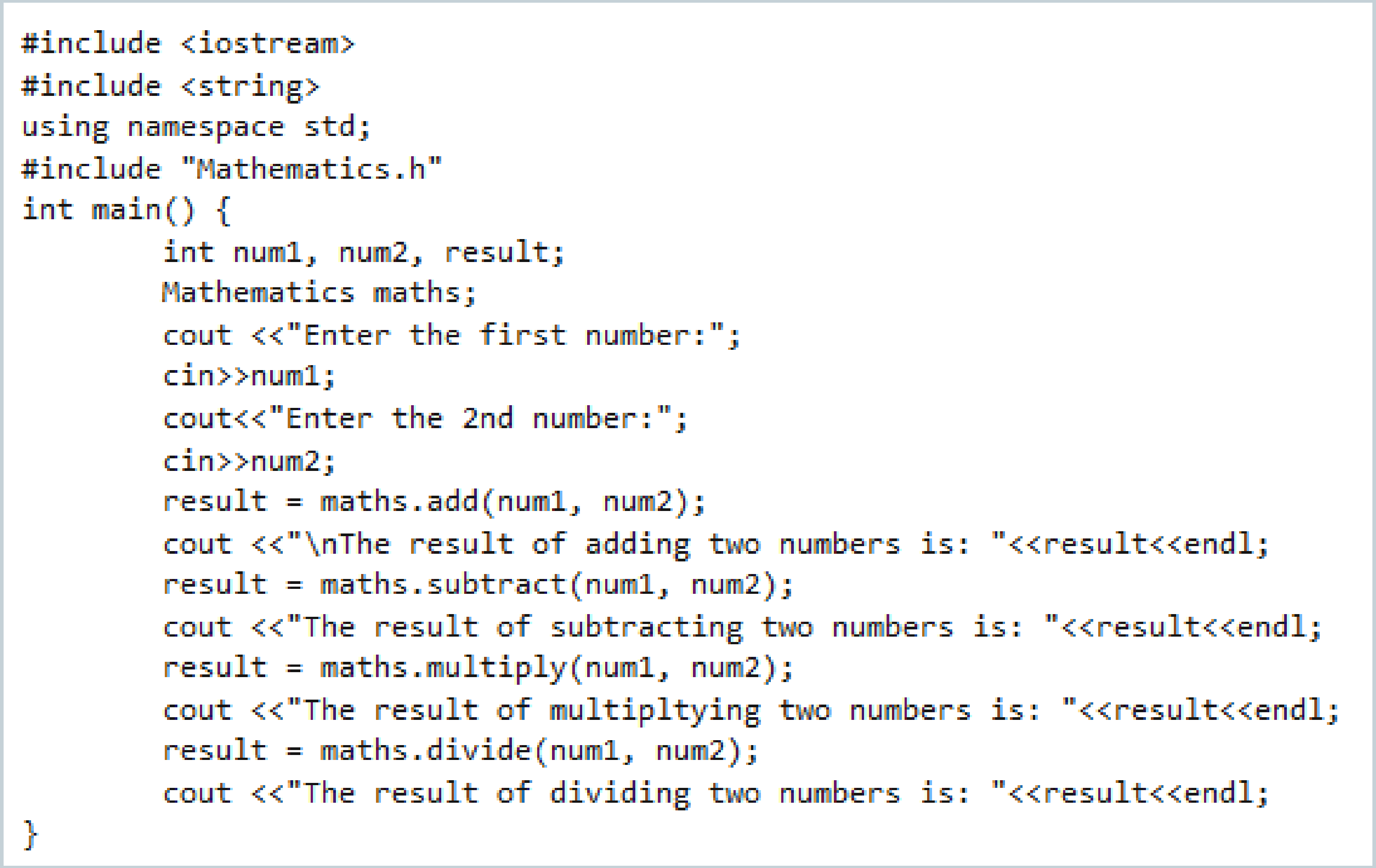
Mathematics.cpp
source file



```
#ifndef MATHEMATICS_H
#define MATHEMATICS_H
#include <iostream>
class Mathematics
{
        int result;
        public:
        int add(int num1,int num2);
        int subtract(int num1,int num2);
        int multiply(int num1,int num2);
        int divide(int num1,int num2);
};
#endif
```

Mathematics.h
header file

Multi-File Programs



```cpp
#include <iostream>
#include <string>
using namespace std;
#include "Mathematics.h"
int main() {
        int num1, num2, result;
        Mathematics maths;
        cout <<"Enter the first number:";
        cin>>num1;
        cout<<"Enter the 2nd number:";
        cin>>num2;
        result = maths.add(num1, num2);
        cout <<"\nThe result of adding two numbers is: "<<result<<endl;
        result = maths.subtract(num1, num2);
        cout <<"The result of subtracting two numbers is: "<<result<<endl;
        result = maths.multiply(num1, num2);
        cout <<"The result of multipltying two numbers is: "<<result<<endl;
        result = maths.divide(num1, num2);
        cout <<"The result of dividing two numbers is: "<<result<<endl;
}
```

Multi-File Programs

- Compiling
  - Let's say you have three .cpp source files, rectangle.cpp, ellipse.cpp, and main.cpp
  - One approach to compilation is:
    - g++ -c rectangle.cpp
    - g++ -c ellipse.cpp
    - g++ -c main.cpp
    - g++ -o myprogram rectangle.o ellipse.o main.o
  - You need not specify the header files because these will be #include(d) in the .cpp files

makefiles

- As you get more and more files, compilation at the command line gets more and more tedious
- You can put your compilation commands into a single file named “makefile” and use the Linux utility, “make” to do the compilation
  - Has similarities to Linux shell scripts
- make program looks at list of requirements in the makefile, checks time stamps, and if something is out of date, re-compiles it
  - That way, only the files that have changed need to be updated

makefiles

- Like shell scripts, you can use variables in a makefile
  - Common variables:
    - CFLAGS = -g -Wall
    - CC = g++
  - To use the variable, use ${varname}
    - e.g. ${CFLAGS}
- You can also insert comments
  - Comments are preceded by the # symbol

makefiles

- Dependencies
  - Rules in a makefile that specify what needs to be done, in what order
    - [name of rule] : [list of other rules, separated by spaces]
      [list of source files, separated by spaces]
    - [TAB] command to execute in the event the rule is violated
  - Called "dependencies" because one rule can depend on the status of another
  - You *must* use a TAB character, not a sequence of spaces, to ensure that your commands will be interpreted correctly
    - You may have multiple tabbed commands to satisfy a rule

makefile Example

- Let’s say we have two files, main.cpp and help.cpp, in our program
- The makefile might look like:

```
main.o: main.cpp

       g++ -c main.cpp

help.o:  help.cpp

       g++ -c help.cpp

main.exe: main.o help.o

       g++ main.o help.o –o main.exe
```

makefiles

- To run a makefile, simply type:
  - make
- make will look for the file called makefile and execute the compilation commands
- If you have several makefiles, you can name them different names (for example MyMakefile) and use the command:
  - make –f MyMakefile
- If you want make to only execute one rule, call that rule:
  - make clean

makefiles – A More Interesting Example

```
# makefile for a frog project

CC=g++
CFLAGS=-g –Wall
RM=rm –f

all: main helloworld

frog.o: frog.h frog.cpp
       ${CC} ${CFLAGS} –c frog.cpp
main: main.o frog.o
       ${CC} ${CFLAGS} –o main main.o frog.o
helloworld: helloworld.cpp
       ${CC} ${CFLAGS} –o helloworld helloworld.cpp
clean:
       ${RM} *.o main
```

makefiles

- Not all of your files need to be in the same directory to be compiled by a makefile
- You can use any Linux command inside a makefile as a command (the tabbed parts)
- You can use make with any compiler – that’s what the CC and CFLAGS variables were about in the last example
- There are dependency generator program that will create makefiles for you if your program is very complex

Summary

- Multi-File Programs
- makefiles