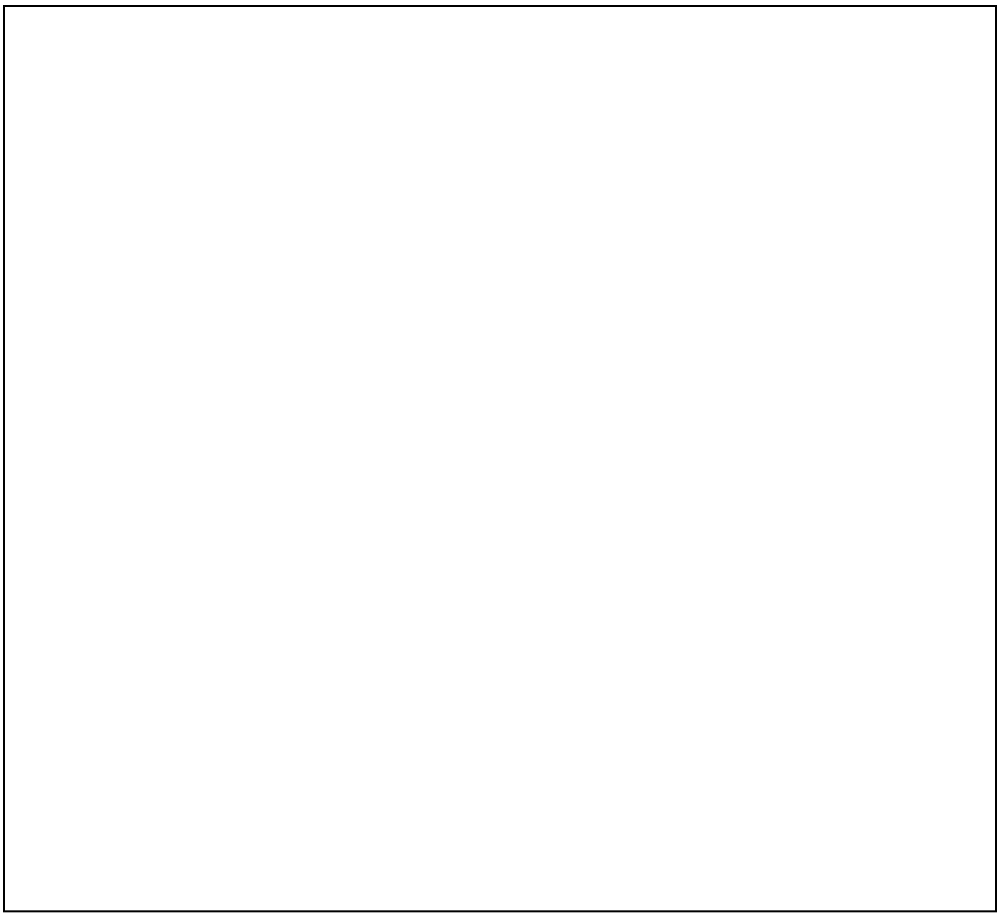
# C++: A simple program

- Intro - I will gloss over a lot of details

## Outline

- Source Code
- Compiling and Running (no IDE)
- Debugging (no IDE)
- IDE and compiler interaction
- Compiling, Running and Debugging with IDE



## Source Code – hello.cpp

```
// a small C++ program
#include <iostream>

int main()
{
    std::cout << "Hello, world!" << std::endl;
    return 0;
}
```

Main special function, you cant call it, may or may not return value (only time compiler will not force return)

all C++ programs start with main(),

libraries Don't have a main

Can also have int main(int argc, char* argv[]) {

Java like stuff ;

- Comments
- case sensitivity
- ;
- {}
- function format
- return

#include – gives declaration of cout, Needed to define what couts params and return type are compiler needs this info in order to set aside memory in .o file for function calls to another file

or library

**open <iostream> in eclipse**

The purpose of a header file is to hold declarations for other files to use.

Std::endl the end of line char

Non Java like

#include

## Something Different– header files

- Java
  - classes are all in 1 file
  - import statements used to include references to classes from libraries
- C++
  - classes are in 2 files (.cpp and .h)
  - Include files reference a library (or object file)- linker includes it in executable
- C++ is more difficult to use in this respect

```
import java.lang.String;
```

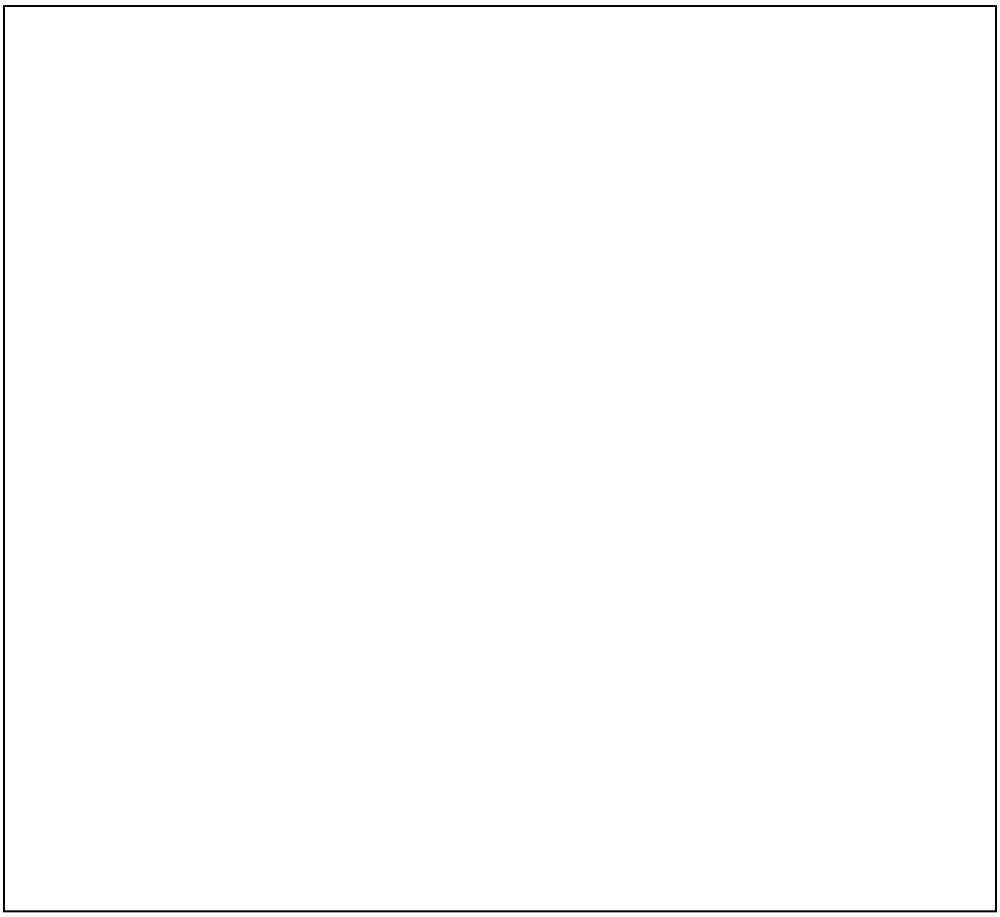
```
#include <string>
```

- C++ is a pain this way, have to manually locate the header files and the .o or library files the headers are associated with
- Takes longer to compile, the preprocessor loads the header, the compiler compiles it and the linker creates an executable

- Libraries
- Libraries are groups of functions that have been “packaged up” for reuse in many different programs. The core C++ language is actually very small and minimalistic — however, C++ comes with a bunch of libraries, known as the C++ standard libraries, that provide programmers with lots of extra functionality. For example, the iostream library contains functions for doing input and output. During the link stage of the compilation process, the libraries from the C++ standard library are the runtime support libraries that are linked into the program

## Outline

- Source Code
- Compiling and Running (no IDE)
- Debugging (no IDE)
- IDE and compiler interaction
- Compiling, Running and Debugging with IDE



# Compilers

- see https://en.wikipedia.org/wiki/List_of_compilers#C.2B.2B_compilers

| Compiler | Author | Windows | Unix-like | Other OSs | License type | IDE? | Standard conformance | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | C++11 | C++14 | C++17 |
| C++Builder | Embarcadero (CodeGear) | Yes | OS X, iOS[2] | No | Proprietary | Yes | Yes/No (Supported via Clang.[3]) | Yes/No | Yes/No |
| Turbo C++ Explorer | Embarcadero (CodeGear) | Yes | No | No | Freeware | Yes | ? | ? | ? |
| C++ Compiler | Embarcadero (CodeGear) | Yes | No | No | Freeware | No | ? | ? | ? |
| CINT | CERN | Yes | Yes | BeBox, DOS, Convex, etc. | X11/MIT | Yes | ? | ? | ? |
| Borland C++ | Borland (CodeGear) | Yes | No | DOS | Proprietary | Yes | No | No | No |
| Turbo C++ for DOS | Borland (CodeGear) | No | No | DOS | Proprietary | Yes | No | No | No |
| Clang | LLVM Project | Yes | Yes | Yes | BSD-like | Xcode, QtCreator (optional) | Yes | Yes | Partial |
| CodeWarrior | Metrowerks | Yes | Yes | Yes | Freeware | Yes | ? | ? | ? |
| Comeau C/C++ | Comeau Computing | Yes | Yes | Yes | Proprietary | No | No | No | No |
| CoSy compiler development system | ACE Associated Compiler Experts | Yes | Yes | No | Proprietary | No | ? | ? | ? |
| Digital Mars | Digital Mars | Yes | No | DOS | Proprietary | No | ? | ? | ? |
| EDGE ARM C/C++ | Mentor Graphics | Yes | Yes | Yes | Proprietary | Yes | ? | ? | ? |
| Edison Design Group | Edison Design Group | Yes | Yes | Yes | Proprietary | No | Yes | Yes | Partial |
| GCC | GNU Project | MinGW, Cygwin | Yes | Yes | GPLv3 | QtCreator, Kdevelop, Eclipse, NetBeans, Code::Blocks, Geany | Yes[4] | Yes | Yes |
| Visual C++ | Microsoft | Yes | can target Linux, OS X, Android and iOS (since VS 2015) | No | Proprietary | Yes | Yes[5] | Yes | Incomplete |

- **Clang was the big dog, now GCC has caught up (most popular on linux) note the IDEs,**
- **Note the state of Visual C++, again its not OS, platform dependant… but it out of the box works**

# Getting a compiler

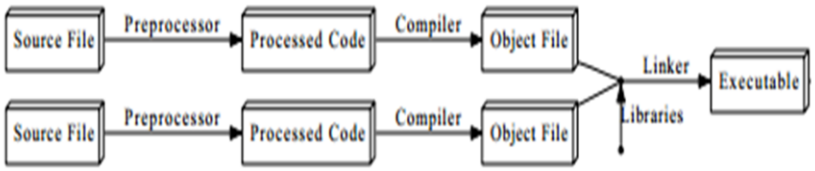
- Visual C++ - comes with MS compiler
- GCC – depends on OS
  - Linux install build essentials to get GCC

```
$ sudo apt-get update
$ sudo apt-get upgrade
$ sudo apt-get install build-essential
$ gcc -v
$ make -v
```

  - Windows – minGW or Cygwin for GCC
    - http://www.mingw.org/wiki/HOWTO_Install_the_MinGW_GCC_Compiler_Suite
    - https://www.cygwin.com/

- **sudo apt-get update $ sudo apt-get upgrade $ sudo apt-get install build-essential**
- **Probably need to add path**
- Edit /etc/bash.bashrc to add the installation path to the PATH environment variable
- **export PATH=/path/bin:${PATH}**
- **Then source bashrc file**
- **Apple- just works**

# Compiling/Linking - overview




**Source File** – .cpp .hpp .h files files
**Preprocessor** – program that performs text substitution
**Compiler**- converts preprocessed source code to object code for a particular processor
**Linker** – Links object files and external libraries to form exe (or library)
Will always link the Cruntime and StandardLibrary

See http://www.ntu.edu.sg/home/ehchua/programming/cpp/gcc_make.html for more information

**Preprocessor** – It works on one C++ source file at a time by replacing #include directives with the content of the respective files (which is usually just declarations), doing replacement of macros (#define), and selecting different portions of text depending of #if, #ifdef and #ifndef directives.

Output is pure C++ no preprocessor directives

Anything with # infront is preprocessor stuff

**Compiler** – generates compiler errors,

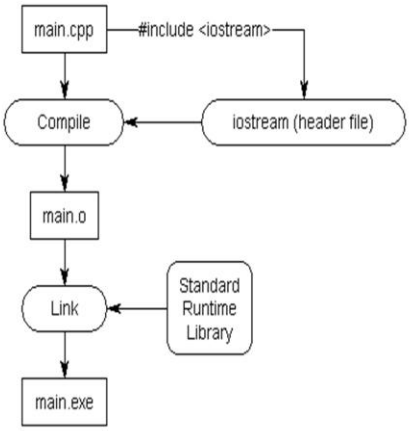
**Linker-** generates linker errors, takes your object files, and any external libraries that you are using, and combines them into an executable (or a library)

**Once again IDE's do this under the covers for you**

Exe or lib that is suitable to run on 1 platform (like windows, sometimes specific versions of windows) or Mac or Linux,

# Compiling/Linking

```
// a small C++ program
#include <iostream>

int main()
{
    std::cout << "Hello, world!" << std::endl;
    return 0;
}
```




See http://www.ntu.edu.sg/home/ehchua/programming/cpp/gcc_make.html for more information
Diagram from http://www.learncpp.com/cpp-tutorial/19-header-files/

Preprocessor reads the #include and inserts that header file into main.cpp (file becomes much larger)

Compile it

Then link it to the standard library (iostream is a description of and how to use all the functions and objects in the standard runtime library) defined in libstdc++-6.dll

Any program that uses <iostream> and compiled with minGW will link to this library

**show example using depends**

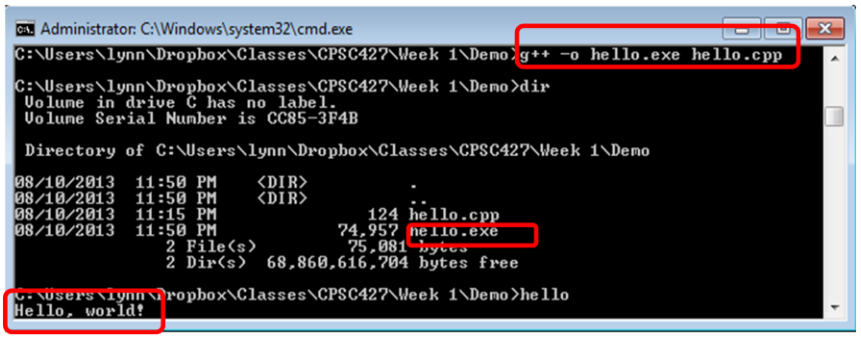
Libstdc++-6.dll standard library for minGW

MSVCRT.dll c runtime and Microsoft

MSVCPRT.LIB  standard library Microsoft

# Compiling/Linking – Example 1

- As simple as g++ -o hello.exe hello.cpp
- Can become very complex
- Commands reside in make file




console apps, no ui, launched from command line,

Build from command line

**-o output executable file**

**-c compile and assemble but no link**

**-g** requests that the compiler and linker generate and retain symbol information in the executable itself.

**-I show include files and where they are from**

**g++ --help**

**Demo this**

# Compiling/Linking – Example 2

- 2 source files; hello.cpp, myfunc.cpp
- 1 user defined header file myfunc.h
- See Project -> 2_files_simple

```cpp
//hello.cpp
#include <iostream>
#include <string.h>
#include "myfunc.h"

int main()
{
    std::string a = myfunc();
    std::cout << a << std::endl;
    return 0;
}
```

```cpp
//myfunc.h
#include <iostream>
std::string myfunc();
```

```cpp
//myfunc.cpp
#include "myfunc.h"

std::string myfunc()
{
    return "hello world";
}
```

**Show proj**

Hello.cpp

**Diff?**

#include <filename>

#include "filename"

difference between is the location the preprocessor searches for the file to be included.

**“”** the preprocessor searches in the same directory as the file being compiled (and also user specified dirs). This method is normally used to include programmer defined headers.

**<>** used for standard library headers - the search is performed in an implementation dependent manner, normally through predesignated directories.

**WHERE ARE THESE INCLUDE FILES?**

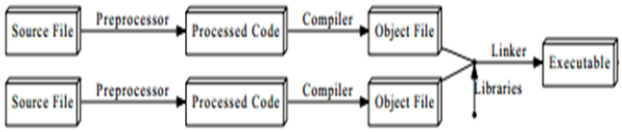
**DEMO**

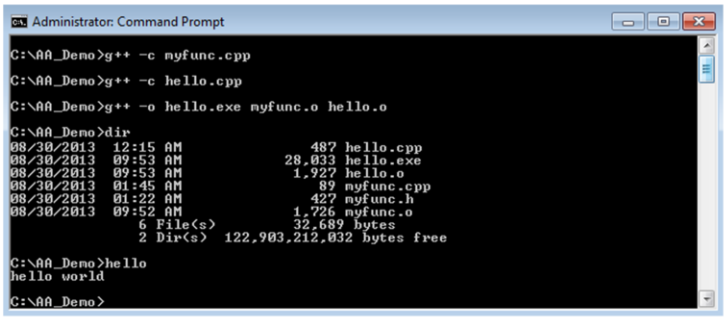
Use ‘include browser’view

Drop hello.cpp on this view

For both <iostream> and “myfunc.h”
Right click->show in->Properties->location

# Compiling/Linking – Example 2




**Slight edit of dir output**

**-c compile and assemble but no link (generate .o object file)**

**-o generate executable from object files**

**g++ --help**

– Here I am compiling 2 files and linking them to form an executable

Preprocessor – subs in contents of <iostream> and myfunc.h into hello.cpp, myfunc.cpp

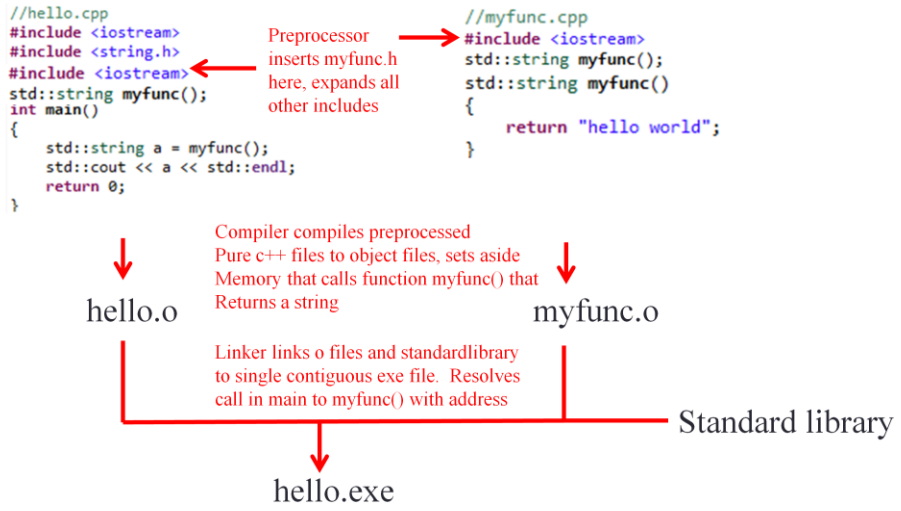
Compiler- converts preprocessed hello.cpp, myfunc.cpp to object code for a *particular processor*

Linker – Links hello.o, myfunc.o and any external libraries to form exe

means 1 continuous block of code or address spaces

**Demo**

# Compiling/Linking – Example 2




Hello uses myfunc(), how does it know definition/declaration?

**CLICK**

Doesn't, until preprocessor replaces #include myfunc.h with declaration

**CLICK**

Now it knows declaration

Compiler sets aside memory needed to call myfunc w/no arguments that returns string (does not know address of call yet)

**CLICK**

Linker wires the 2 object files and the standard library into a single exe

Resolves call to myfunc with an absolute address

How does it do this? Demo on board

# Compiling/Linking – Example 2

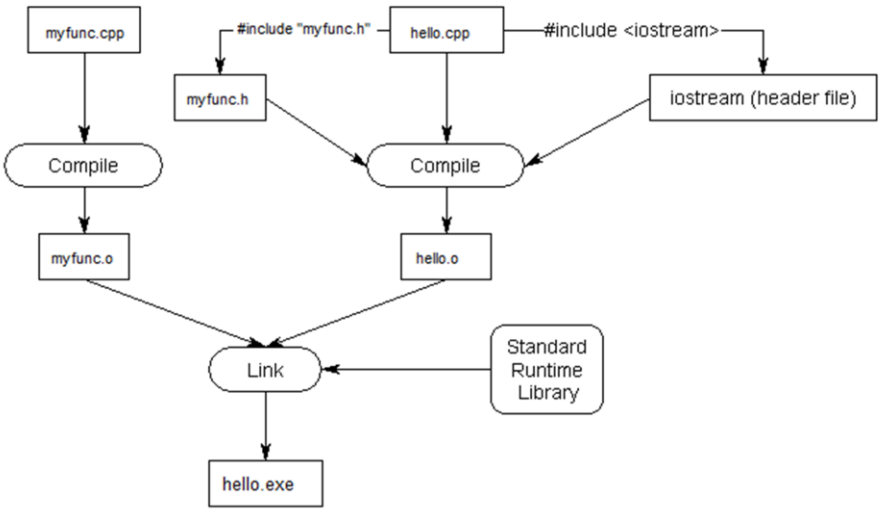



Diagram from http://www.learncpp.com/cpp-tutorial/19-header-files/

Diagram – if cant do prev slide animation

Point

Most projects have many files, compilation of each file takes time

**If just change 1 file why recompile all**? If cpp file **just recompile that file and relink all** the other object files

Saves a lot of time, Eclipse does not do this, it will rebuild the whole thing

Visual studio does do this, its better than eclipse in this respect for large projects

# Makefiles – a way to automate things

```
1  #target exe
2  myexe: hello.o myfunc.o
3      g++ $(CFLAGS) -o myexe hello.o myfunc.o
4
5  #rebuild if either of the files below change
6  hello.o: hello.cpp myfunc.h
7      g++ $(CFLAGS) -c hello.cpp
8
9  #rebuild if either of the files below change
10 myfunc.o: myfunc.cpp myfunc.h
11     g++ $(CFLAGS) -c myfunc.cpp
12
13 #type 'make clean' to remove following
14 clean:
15     rm -f *.o myexe.exe
```

C:\test\myfunc.cpp

This object file depends on these two source files, if eithe change rebuild the object file

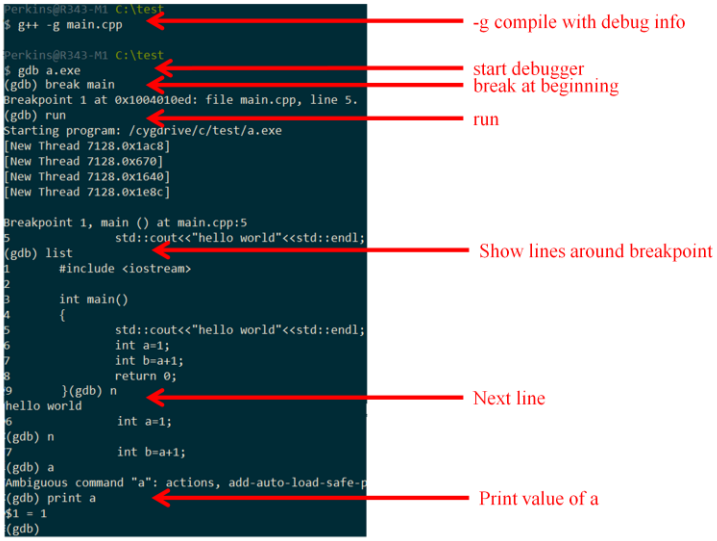
```
cmd (Admin)
<1> cmd
Perkins@R343-M1 C:\test
$ make clean
rm -f *.o myexe.exe

Perkins@R343-M1 C:\test
$ make
g++  -c hello.cpp
g++  -c myfunc.cpp
g++  -o myexe hello.o myfunc.o

Perkins@R343-M1 C:\test
$ myexe
hello world

Perkins@R343-M1 C:\test
$
```

- **Save as allfiles  makefile**

## Outline

- Source Code
- Compiling and Running (no IDE)
- Debugging (no IDE)
- IDE and compiler interaction
- Compiling, Running and Debugging with IDE

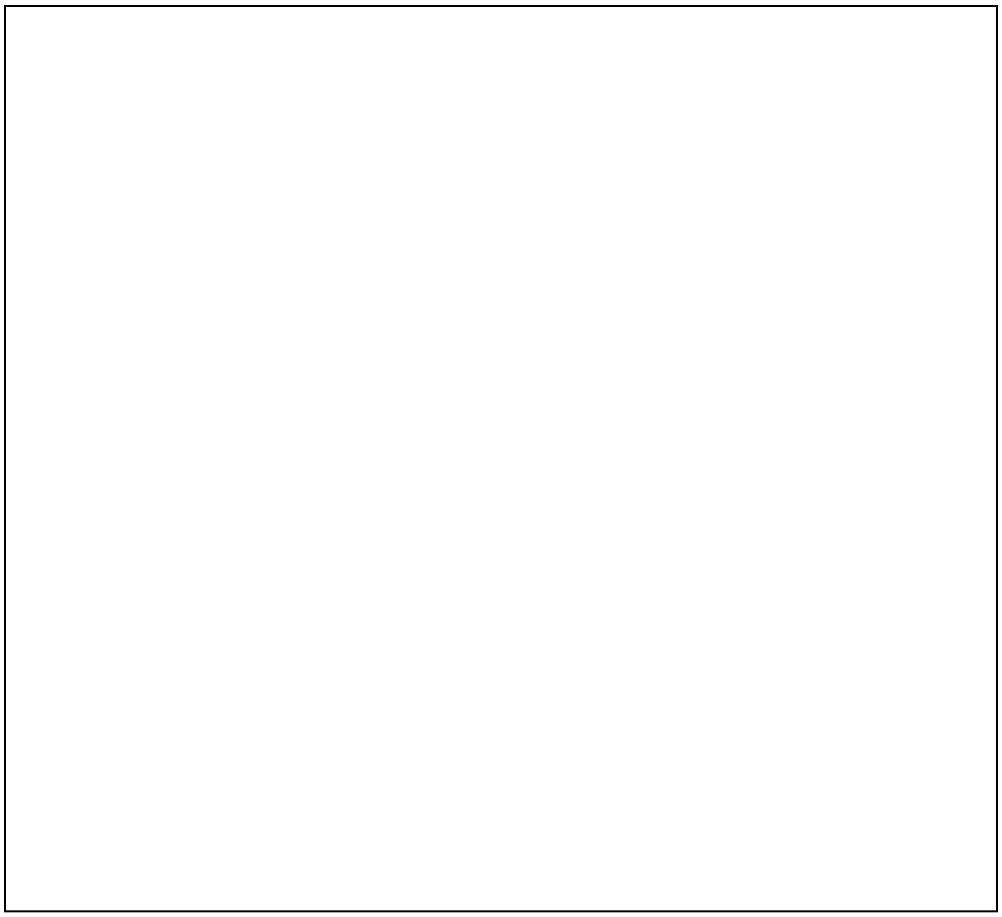
# Debugging



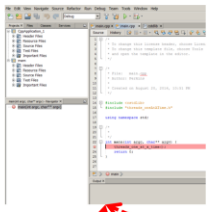

- **https://www.cs.swarthmore.edu/~newhall/unixhelp/howto_gdb.html#badprog**

## Outline

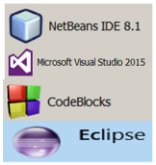
- Source Code
- Compiling and Running (no IDE)
- Debugging (no IDE)
- IDE and compiler interaction
- Compiling, Running and Debugging with IDE



# IDE and compiler interaction



Integrated Development environment (IDE)
Such as…




But an IDE makes it easier
Especially on large projects

IDE uses compiler

IDE uses debugger

Compiler (like gcc)

Debugger (like gdb)

You only need these

debugs

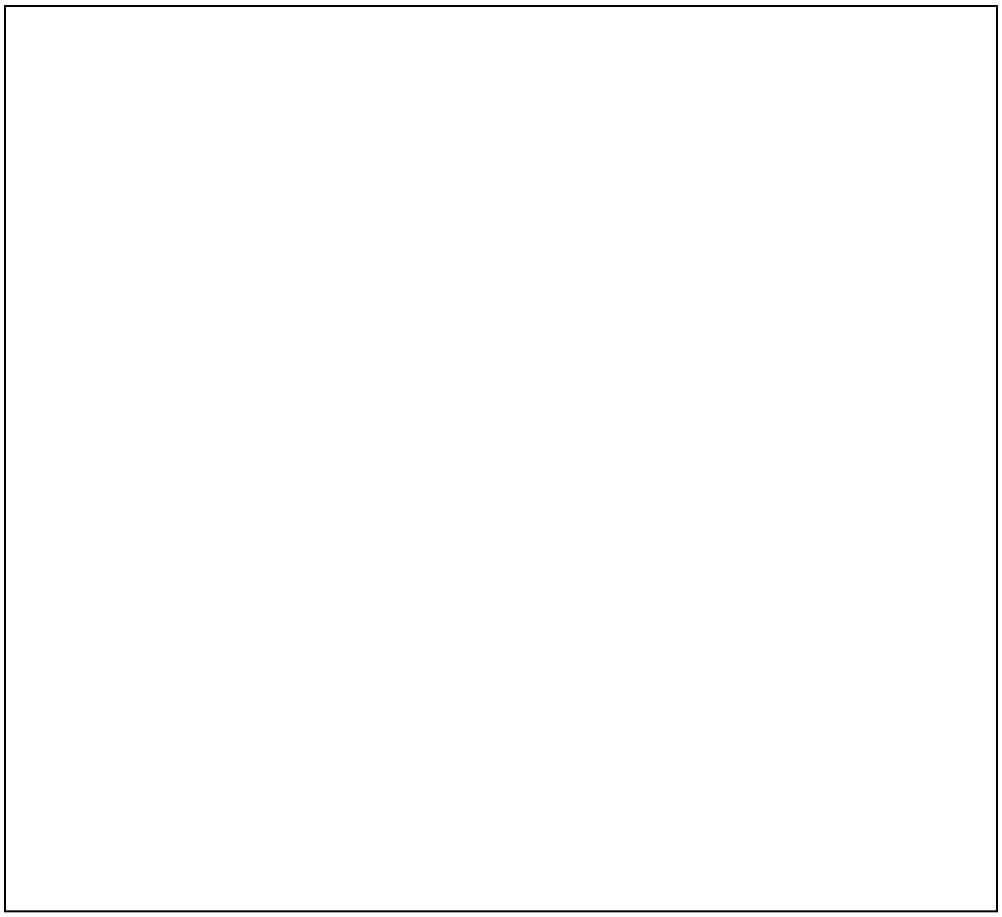
Compiler generates executable

Executable application

To generate this

## Outline

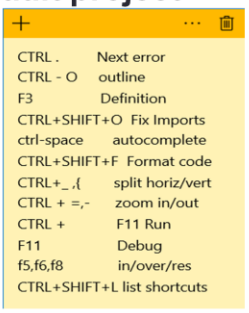
- Source Code
- Compiling and Running (no IDE)
- Debugging (no IDE)
- IDE and compiler interaction
- Compiling, Running and Debugging with IDE



# Compiling/Linking – Using an IDE

- **Let Integrated Development Environment (IDE) handle all details**
- **(build settings still there just using default project settings)**

- Create C++ project
- Copy 3 files from example 2 to it
- Build it
- Here are some key shortcuts




Key bindings I use

Know about make process (Compile/Link,Load on cmd ) but don't have to be expert

Can set various compiler and linker switches and options in Eclipse, do this as your projects become more sophisticated

Demo

Go over what is in directory

Include folders, declarations of stuff we will need

<IOStream> put mouse on it and F3 to see its contents

Where is it, (save as does not work correctly in eclipse)

Add the 'include browser' (window->show view->C++ …)

drop file on it, list all the includes

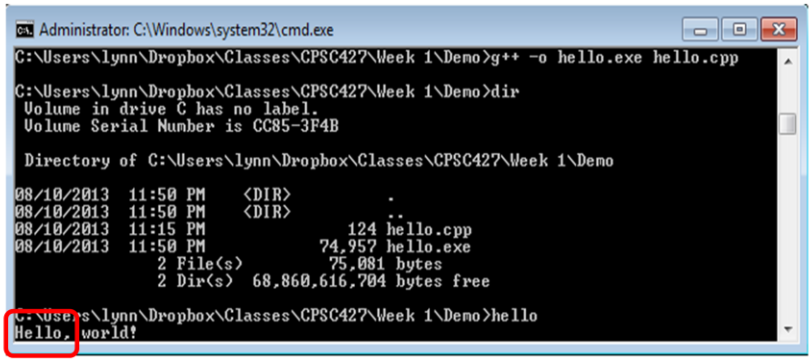
right click on <iostream>, Show in->properties

See where its coming from (helps to prevent insidious bug where you are including stuff from different versions

minGW4.6 verses 4.8 etc)

# Running

- Its an Executable! (no virtual machine)
- Can run from command line or IDE
- Fast Demo Various bits of IDE




Run it,

from eclipse will be in the console window

from file system it will be in a command window

When only Cout from program then exit, doubleclick on .exe will cause cmd window to open cout then close, very quickly

Instead open command window, go to debug dir, and run, output will stay visible

**show example using depends**

Show dlls attached to it as part of process explorer, or dependency walker

useful for seeing if wrong version of dll is attached (dll hell, can't tell which version of dll is attached )

Libstdc++-6.dll standard library for minGW

MSVCRT.dll c runtime and Microsoft

MSVCPRT.LIB standard library Microsoft

We will go over all these techniques as we go along

## What have we learned

- C++ has lots of similarities to Java (more as we go)
- How to write a simple C++ program
- How to compile using command line
- How to use an IDE to create a program
- **For this class and most likely professionally, let the IDE manage your builds.**
- Basic IDE usage (Debug/release build, variables, breakpoints etc)
- How to run a program
- PRACTICE PLEASE

**However, open source people often really know the build process, the language and have efficient development techniques.**

**Finished 3-4:15 class**

**Must have a main**

**Faster than java**

**Alluded to standard library**