

## Lecture 13 CSI333

Slide 1

file **Anthem.C**:

```
#include "OhSay"  
#include "WhatSo"  
#include "WhoseBroad"  
And the rockets red glare  
The bombs bursting in air  
Gave proof through the night  
That our flag was still there  
Oh say does that star spangled banner yet wave  
Ore the land of the free and the home of the brave?
```

## Slide 2

file **OhSay:**

```
Oh say can you see  
By the dawns early light
```

file **WhatSo:**

```
What so proudly we hailed  
At the twilights last gleaming.
```

file **WhoseBroad:**

```
Whose broad strips and bright stars  
Through the perilous fight  
Ore the ramparts we watched  
Were so gallently streaming?
```

Command:

```
$ g++ -E -P Anthem.C      # Stop after Preprocess Step
```

**Slide 3**

Oh say can you see  
By the dawns early light

What so proudly we hailed  
At the twilights last gleaming.

Whose broad strips and bright stars  
Through the perilous fight  
Ore the ramparts we watched  
Were so gallantly streaming?

And the rockets red glare  
The bombs bursting in air  
Gave proof through the night  
That our flag was still there  
Oh say does that star spangled banner yet wave  
Ore the land of the free and the home of the brave?

How the Preprocessor Handles #include Statements

The **first step** `g++ -c someth.C` does is

- Scan to find and **CUT** the first `#include` statement.
- **PASTE** the **CONTENTS** of the included file in its place.
- Resume this scanning and substitution processing from the beginning of the **PASTED** text.

(This is one of several cases of **preprocessor** operations performed during the preprocessor's scan. Others are **macro definition**, **conditional compilation** on macro values, **macro substitution**, ... Macro substitution is valuable in **C**, but is somewhat obsolete in **C++**.)

Slide 4

**Slide 5**

Readings from Stroustrup on Modularization/Separate Compilation: 2.4(intro), 2.4.1, 8.1, 9.1, 9.2.1, 9.2.3, 9.3 (skip namespace stuff)

Readings from Stallman/McGrath et. al. on gmake:  
Follow directions on Lab 3 assignment; also available from <http://www.cygnus.com/pubs/gnupro/>

Preprocessor: H/S All of Chapter 3

Stroustrup 7.8, 9.2.1, 9.3.3, A.11

Bookmark <http://www.cygnus.com/pubs/gnupro/> for future reference.

## Slide 6

```
// strread.h : INTERFACE of strread module
extern char * strread(void);
// strread():
// (1) Returns 0 if standard input is at EOF.
// (2) Exits with a message if the next line
//     read from standard input is too long.
// (3) Otherwise, it returns a pointer to a
//     free-store C-string that contains the
//     next line from standard input,
//     without the \n.
```

What `extern` means: The name declared here signifies an object that will be **DEFINED SOMEWHERE ELSE**.

Prototype Declaration: Specifies types of return val. and arguments. `(void)` means 0 arguments; `()` means NOT A PROTOTYPE.

## Slide 7

```
#include <iostream.h>
#include "strread.h"
// "include" INTERFACE to strread

// main.cpp : Test driver for strread module.

int main( int argc, char * argv[] )
{
    char *pch;
    while( pch = strread() )
        { // strread() was successful.
            cout << pch << endl;
            delete pch; // Reclaim free-store storage
        }
    cout << "strread returned 0, bye.." << endl;
    return 0;          }
}
```

## Slide 8

```
// strread.cpp : IMPLEMENTATION of module strread
#include <iostream.h>
// strread USES iostream (standard C++ library)
#include <string.h>
// strread USES C strings (standard C/C++ library)
#include "strread.h"
// "Include" a copy of our INTERFACE, so the compiler
// will check that our IMPLEMENTATION is consistent
// with the INTERFACE.
static const int BUFSIZE = 40; //private
char * strread(void)
{
    char *pch;
    char buffer[BUFSIZE];
    if( ! cin.getline( buffer, BUFSIZE ) )
        {

```

## Slide 9

```

// ! (stream) returns true if ios::fail is true
// Something's wrong..
if( cin.eof() )
{
    // END OF FILE, normal end of input.
    return 0;
}
// Otherwise, newline was not found among the
// up to BUFSIZE characters examined by getline.
cout << "input too long." << endl;
exit( 1 ); // Bail out; foil "crackers".
} // cin.getline was successful.
pch = new char[strlen(buffer) + 1];
strcpy( pch, buffer ); // Trust getline
// did put a terminating \0 into buffer[].
return pch; }

```

## Slide 10

```

g++ -c main.cpp (what compiler sees to produce main.o)
...LOTS of iostream stuff omitted...
// from streadd.h
extern char * streadd( void );
// this DECLARATION did NOT DEFINE streadd

// from main.cpp
int main( ... )
{
    char *pch;
    while( pch = streadd() )
    The compiler checks streadd takes 0 arguments
    and returns char *
    ...
}

```

## Slide 11

```

g++ -c stread.cpp (what compiler sees to produce stread.o)

...LOTS of iostream & string stuff omitted...

// from stread.h
extern char * stread( void );
// this DECLARATION did NOT DEFINE stread

char * stread( void ) //DEFINE stread here
//The compiler checks that this defines a function
//with 0 arguments and return type char *
{ body of stread
  char *pch;
  ... Compiled into MACHINE INSTRUCTIONS
  return pch;
}

```

## Slide 12

```

When I replaced
while( pch = stread( ) ) by
while( pch = stread( 3 ) )
in main.cpp, the compiler complained:

$ make
g++ -g -c main.cpp
stread.h: In function 'int main(int, char **)':
stread.h:3: too many arguments to function 'stread()'
main.cpp:10: at this point in file
make: *** [main.o] Error 1

$

```

## Slide 13

Executable file Build Steps

The command `g++ simple.cpp` makes this happen:

1. Preprocess: Do inclusions, macro substitution, etc.
2. Compile: “Compiler Proper” transforms preprocessed C++ into **Assembly Language**. Output is named a “.s file”
3. Assemble: An **Assembler** assembles assembly language into an **object file**, which contains **machine instructions** and some **undefined symbols**. Output is named a “.o file”
4. Link: A **Linker** combines your object file with library object files (which implement `readline`, `new`, etc) to build an **executable** file. Output is named with “.exe”, or with no extension in Unix.

The GNU compiler’s `-v` (lower case) option shows this.

Slide 14

```
Command:(blank lines removed)
$ g++ -E Anthem.C

# 1 "Anthem.C"
# 1 "OhSay" 1
Oh say can you see
By the dawns early light
# 1 "Anthem.C" 2
# 1 "WhatSo" 1
What so proudly we hailed
At the twilights last gleaming.
# 2 "Anthem.C" 2
# 1 "WhoseBroad" 1
Whose broad strips and bright stars
Through the perilous fight
Ore the ramparts we watched
Were so gallently streaming?
# 3 "Anthem.C" 2
And the rockets red glare
The bombs bursting in air
Gave proof through the night
That our flag was still there
Oh say does that star spangled banner yet wave
Ore the land of the free and the home of the brave?
```

## Slide 15

The most important reasons (today) for **SEPARATE COMPILATION** of **MULTIPLE** .cpp files with some common **header** files are:

- Non-trivial software is best designed as multiple **modules** each solving a separate problem. Modules **USE** one another.
- Distinct **header** and **implementation** files **SEPARATE** the **interface** from the **implementation** of each module, in a way that helps guarantee the implementation and all **USES** of the interface are consistent.

Keep **EXACTLY ONE** copy of the interface, **NEVER** copy it manually, so copying or version change errors **NEVER** happen!

**Slide 16**

Other reasons for separate compilations:

- It can make the rebuilds required after bug fixes and modifications much faster.
- It's easier and faster to write, edit and compile (for syntax checking) a short file than a loooooooooooooooooooooong (long) file.
- It helps keep names private to one module from interfering with names private to others (especially for non-object oriented C/C++ projects).
- It facilitates code sharing and reuse.
- It helps different people to simultaneously work on different modules for a large project.
- It enables proprietary (secret) implementations to have published interfaces.

**Slide 17**

“Make” is a tool that **AUTOMATES** the generation of the commands (like `g++ -c something.cpp` and `g++ A.o B.o ...`) to perform software build steps.

(If used properly), make will **ONLY** perform the **steps** made **NECESSARY** because of input file modifications or missing output files.

For example, if you modify just `main.cpp`, both `main.o` and `main` must be rebuilt, **BUT NOT** `strread.o`.

**Slide 18**

File X **directly depends on** file Y means file Y is an **input** to the operation (compile, link, etc.) that **outputs** file X.

Therefore, if file Y is edited, file X becomes **out-of-date** and must be deleted and rebuilt.

Suppose `main.cpp` “includes” `strread.h`  
`main.cpp` does NOT DEPEND ON `strread.h`  
`main.o` and (indirectly) `main` do.

X=`main.o`

Y=`strread.h` or Y=`main.cpp` (X depends on Y)

## Slide 19

Sample Makefile (named Makefile, read and analyzed when you command `make` or `gmake`)

```
main : main.o strread.o
      g++ -o main main.o strread.o
```

```
main.o : main.cpp strread.h
      g++ -g -c main.cpp
```

```
strread.o : strread.cpp strread.h
      g++ -g -c strread.cpp
```

Form of a rule:

```
TARGET : DEPENDENCIES  
(invisible TAB char)COMMAND1  
(invisible TAB char)COMMAND2 etc..
```