CS197c: Programming in C++

Lecture 1

Marc Cartright

http://ciir.cs.umass.edu/~irmarc/cs197C/index.html

Administration : course details

- Number of credits : 1 (no audit allowed)
- Graded class (you must choose Pass/Fail)
- Class time : Wednesdays 3:35 – 5:00 pm
- Course webpage
  - http://ciir.cs.umass.edu/~irmarc/cs197c/index.html
- Office hours
  - Thursdays, 10am-12pm
  - If you *cannot* make that time, email me to work something out.

Administration : prerequisites

- Should have prior programming experience
  - C or JAVA or Python
- Will need to have basic knowledge of UNIX/ Linux OS
  - Must be able to work on edlab machines

Administration : reading material

- References:
  - Lots online
  - Lecture notes
  - Examples

- Recommended books:
  - C++ for Java programmers, Mark Weiss
  - Practical C++, Robert W. McGregor
  - C++ Primer, Stanley B. Lippman

Administration : grading

- 6 Short Quizzes
  - Assigned after lecture, due that Friday
  - Total of 15% of final grade.
- 4 Programming Assignments
  - 2 weeks for each assignment = NO EXCUSES
  - Each is 20% of final grade
- 5% for participation/effort
- Talking is ok, but you must program alone
- No late days

Syllabus

- Lecture 1 : C/C++ basics, tools, Makefiles, C data types, ADTs
- Lecture 2 : C libraries
- Lecture 3 : Classes in C++, C++ I/O
- Lecture 4 : Memory & Pointers
- Lecture 5 : More Pointers
- Lecture 6 : Templates and the STL
- Lecture 7 : Reflection, Exceptions, C++11
- Lecture 8 : Adv. Topics: Boost and OpenGL

Syllabus: Programming

- Lecture 1 : C/C++ basics, tools, Makefiles, C data types, ADTs
- Lecture 2 : C libraries
- Lecture 3 : Classes in C++, C++ I/O
- Lecture 4 : Memory & Pointers
- Lecture 5 : More Pointers
- Lecture 6 : Templates and the STL
- Lecture 7 : Reflection, Exceptions, C++11
- Lecture 8 : Adv. Topics: Boost and OpenGL

Syllabus: Quizzes

- Lecture 1 : C/C++ basics, tools, Makefiles, C data types, ADTs
- Lecture 2 : C libraries
- Lecture 3 : Classes in C++, C++ I/O
- Lecture 4 : Memory & Pointers
- Lecture 5 : More Pointers
- Lecture 6 : Templates and the STL
- Lecture 7 : Reflection, Exceptions, C++11
- Lecture 8 : Adv. Topics: Boost and OpenGL

Administration : Assignments

- P1
  - Purpose: establish workflow, practice with only C
  - Series of small programming tasks
  - Lots of reading/writing using functions
  - Learn about the C Library

Administration : Assignments

- P2
  - Purpose: Dealing with Classes and Pointers
  - Implement a doubly-linked list w/ pointers
  - Above and Beyond: add iterator, reverse_iterator

Administration : Assignments

- P3
  - Purpose: Getting used to templates
  - Extending the Linked List as a templated class
  - Using the STL

Administration : Assignments

- P4
  - Purpose: Putting together templates, classes, and pointers
  - Implement some of a Red-Black Tree
  - Above and Beyond: Do a B-Tree

Administration : Policies

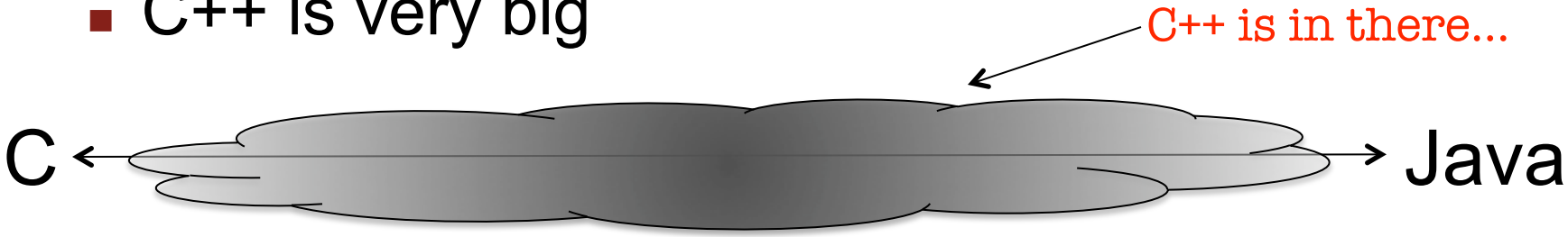
- I will provide: Makefiles, Header files, Test Cases
- You provide: Code that compiles on an EDLAB machine using the makefile and header file
- If it does not compile, it will not be graded.
  - **no exceptions**
  - whining and/or excuses are examples of what will drop your participation grade
- If it is late, it will not be graded.

Administration : Policies

- Cheating/Plagiarism is not tolerated
  - Offending assignment will be given 0
  - Participation component of class dropped to 0 instantly
  - Incident will be escalated to university officials for disciplinary action
  - Guide on academic honesty:
    http://www.umass.edu/dean_students/codeofconduct/acadhonesty/

What is C++ ?

- Originally (C with classes)
- Designed to handle larger code bases
- Characteristics
  - Low-level (memory management exposed)
  - Has object support (classes etc.)
  - Strongly typed (types unambiguous)
  - C++ is very big



C++ and C

- C: the canonical systems language
  - Developed in 1972
- C is C++'s daddy
  - Developed in 1979, by Bjarne Stroustrup
  - (Almost) all valid C code is valid C++
  - C++ designed as a superset of C
  - "C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off"

C++ and Java

- C++ is Java's crazy uncle
- Some similarities
  - Java based on some C++ syntax
  - Many object-oriented features shared
- Some differences
  - C++ has no runtime checks
  - C++ does not have a virtual machine
- Will feel like we're going backwards, because we are

How is C++ going to help you ?

- Will help you write operating systems
  - Earn a good grade in 230 (or equivalent)
- Gives better appreciation of Java
- Helps you write various applications:
  - Games like World of Warcraft
  - Firefox
  - MATLAB, and more...
- Helps you get a job
  - e.g. most Microsoft software is written in C++

Before C++ : Tools

- Compiling tools: *make* and *g++*
- Text editors : *vim* or *emacs* or anything you like
- Debugger : *gdb*
- Basic Unix:

  cd, ls, rm, less, cp, mv, ssh, scp, man

Tools : make and g++

- Make : used to build executables from source code
  - Handles compiling and linking
  - Similar to *ant* for Java

- g++ : the gnu C++ compiler
  - Large number of options (-g , -o, -c, -E, -v)
  - Outputs an executable for that operating system

Tools : text editors

- Good text editors can make life easier
  - Syntax highlighting
  - Parenthesis matching
- Recommended editors
  - emacs (my preference)
  - vim
- On your own
  - Visual Studio, NetBeans, Eclipse...

Tools : gdb debugger

- The gnu debugger
  - Examine live programs
    - Watch variables and memory
    - Start and stop control flow
    - Set break points

Tools : Unix commands

- cd : change directory
- ls : see directory contents
- rm : delete a file
- less : read a text file
- cp : copy a file
- mv : rename or move a file
- ssh : secure shell login
- scp : secure remote copy
- man : read about a command

Windows and Mac Tools

Windows

- PuTTY or SecureCRT
- WinSCP
- XEmacs, Emacs, Vim

Mac

- Terminal, XTerm
- Unix cmds:
  - ssh
  - scp, sftp
  - emacs,vim,xemacs

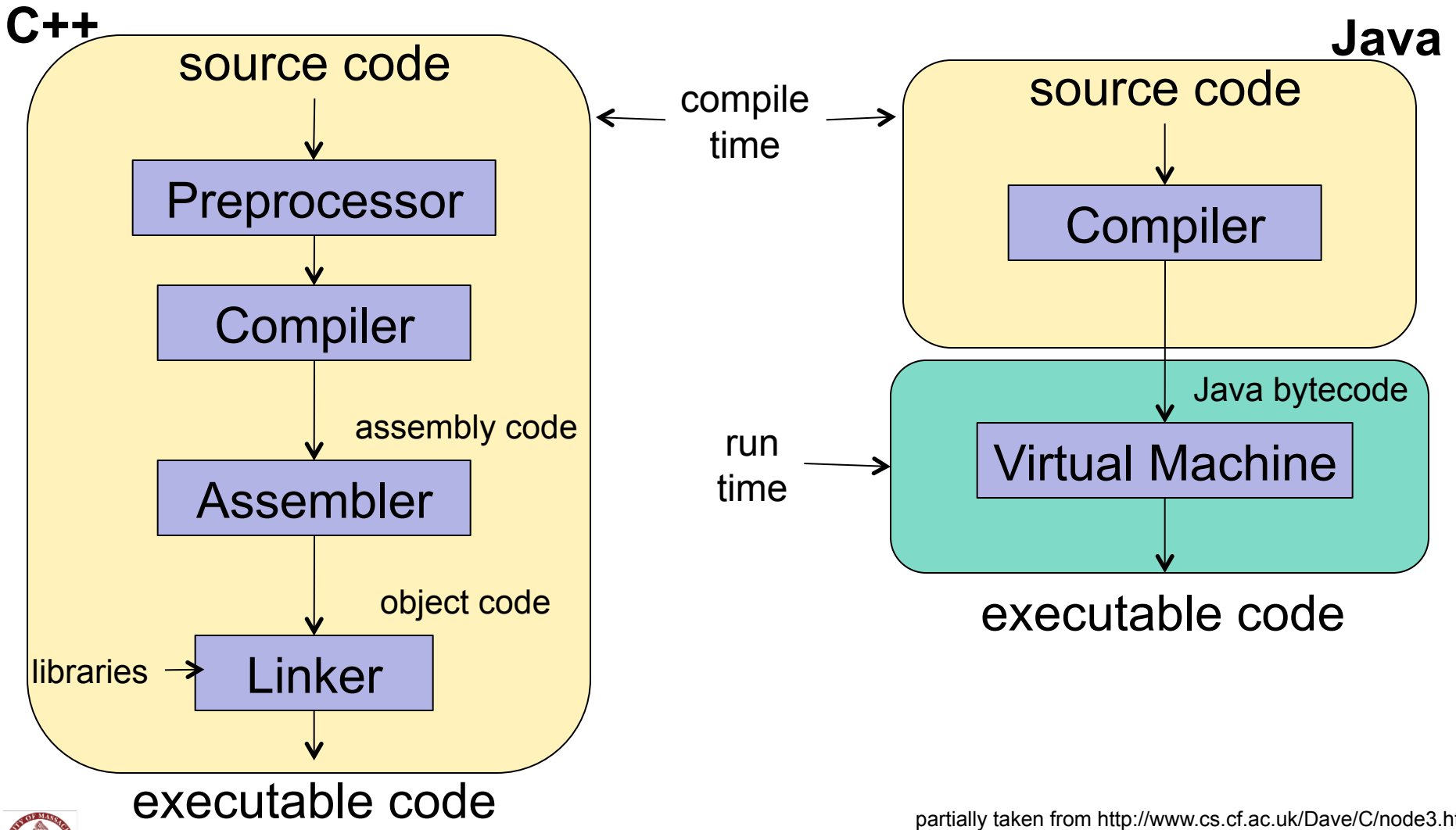
I will not help you install or setup anything for class.

To See More...

- Examples on class website (see first slide)
  - use of various concepts
  - quick tutorial on Unix commands
  - use of some tools

Compilation Models

- Source code → machine executable code



partially taken from http://www.cs.cf.ac.uk/Dave/C/node3.html

Preprocessor Directives (some...)

- #include
  - drags in header files
- #define, #undef
  - C-style macros
  - constants
  - predecessor to inlining
- #ifdef
- #ifndef
- #else
- #endif

Preprocessor version of if/else

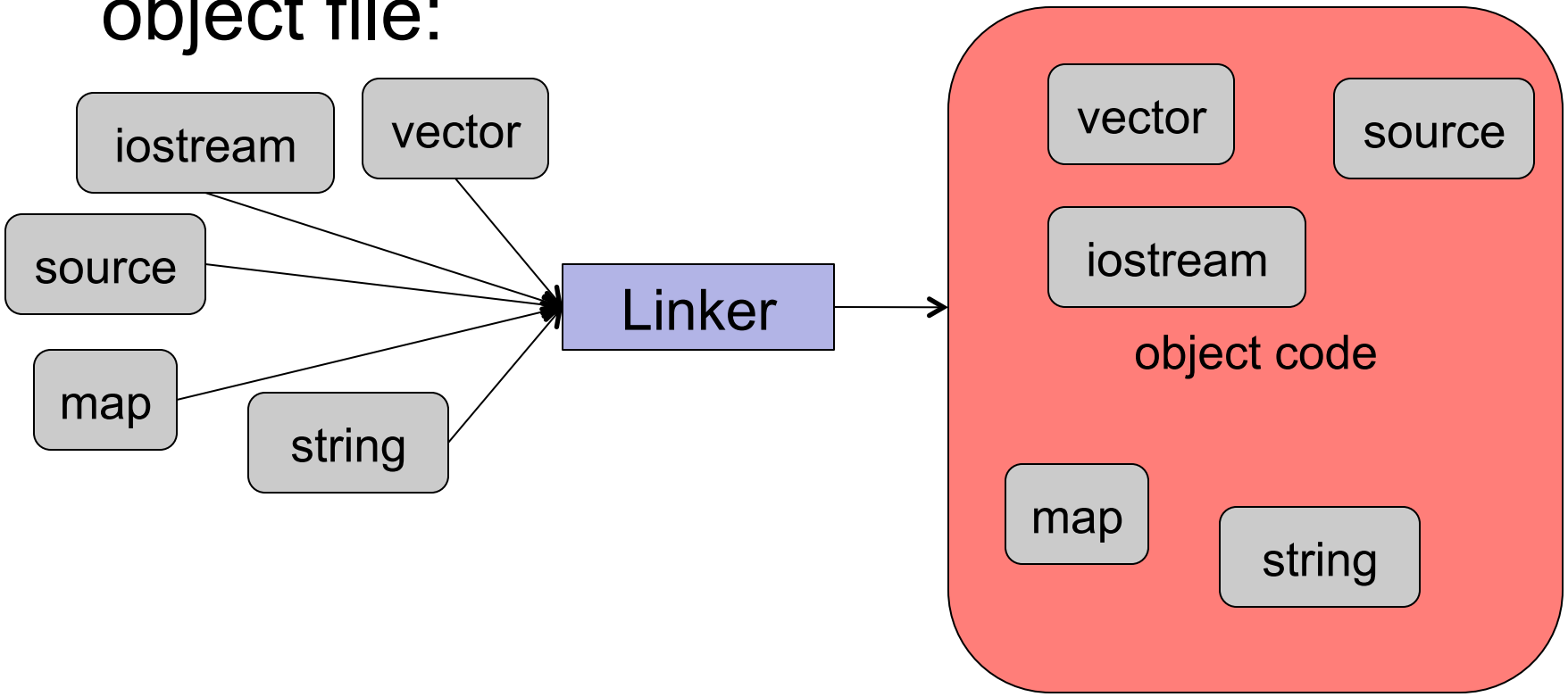
C++ Compilation Model: Linking

- "Linking" puts together separate object files to make final output (done by *link editor)*
  - program
  - another object file

- 2 types of linking: *static* and *dynamic*

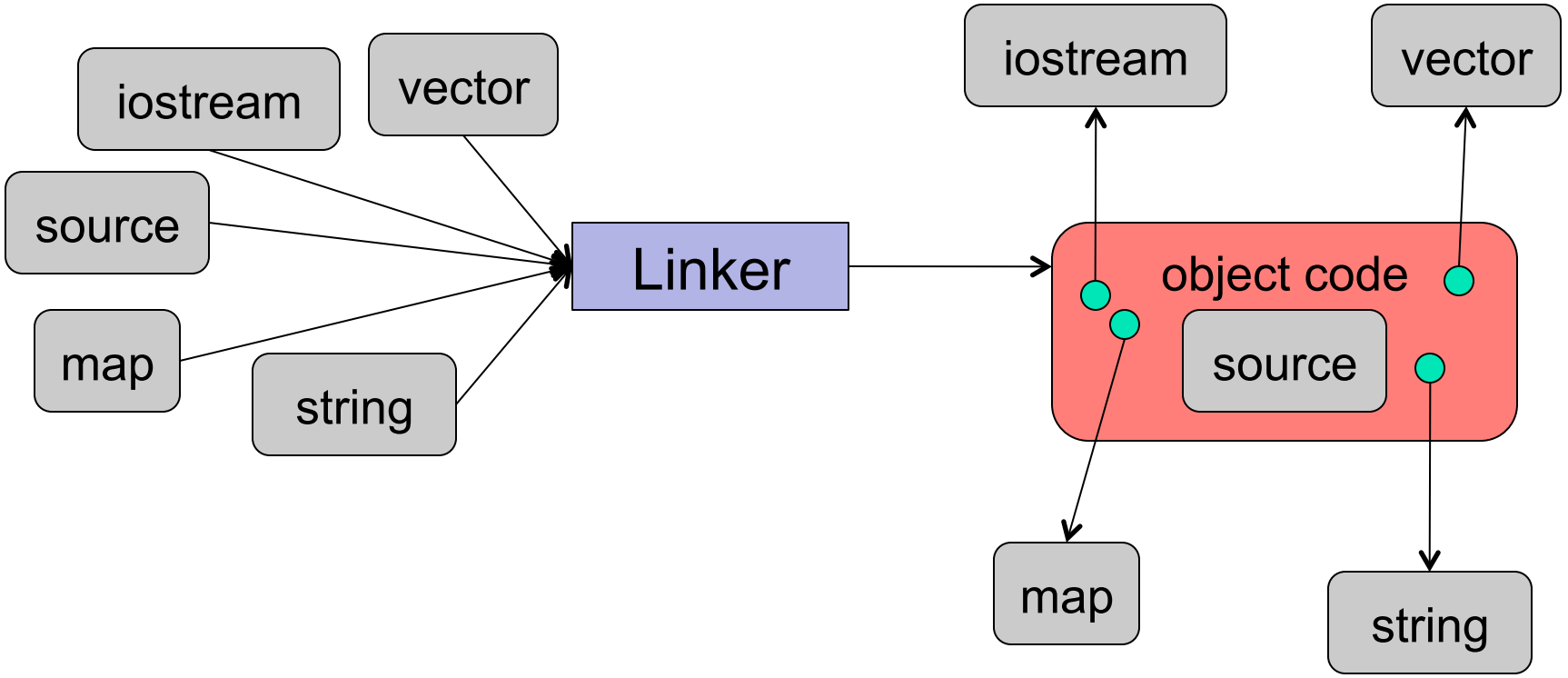
Static Linking

- All included libraries are copied into the object file:



Dynamic Linking

- Object file contains pointers to libraries, resolves at runtime:



Linking Comparison

- Static Linking
  - No runtime resolving needed (speed)
  - Dependencies removed
- Dynamic Linking
  - Reuses code
  - Keeps object files small
  - Better versioning control

Makefiles

```
CC=g++

all:    main

main: main.cpp Parser.o
        ${CC} -I. -L. -o parser main.cpp Parser.o

clean:
        rm -f parser *.o *~
```

Makefiles

```
CC=g++

all:    main

main: main.cpp Parser.o
        ${CC} -I. -L. -o parser main.cpp Parser.o

clean:
        rm -f parser *.o *~
```

Makefiles: Setting variables

```
CC=g++
```

Allows a variable, but just does string replacement:

```
<variable>=<value>
```

When using the variable:

```
${<variable>}, e.g. ${CC}
```

Makefiles

```
CC=g++

all:    main

main: main.cpp Parser.o
        ${CC} -I. -L. -o parser main.cpp Parser.o

clean:
        rm -f parser *.o *~
```

Makefiles: Setting targets

```
all:    main
```

- Targets are used to manage build configurations:

```
<target>:    <dependencies>
             <command(s)>
```

- Dependencies can be other targets, source files, object files...
- Need only dependencies or commands

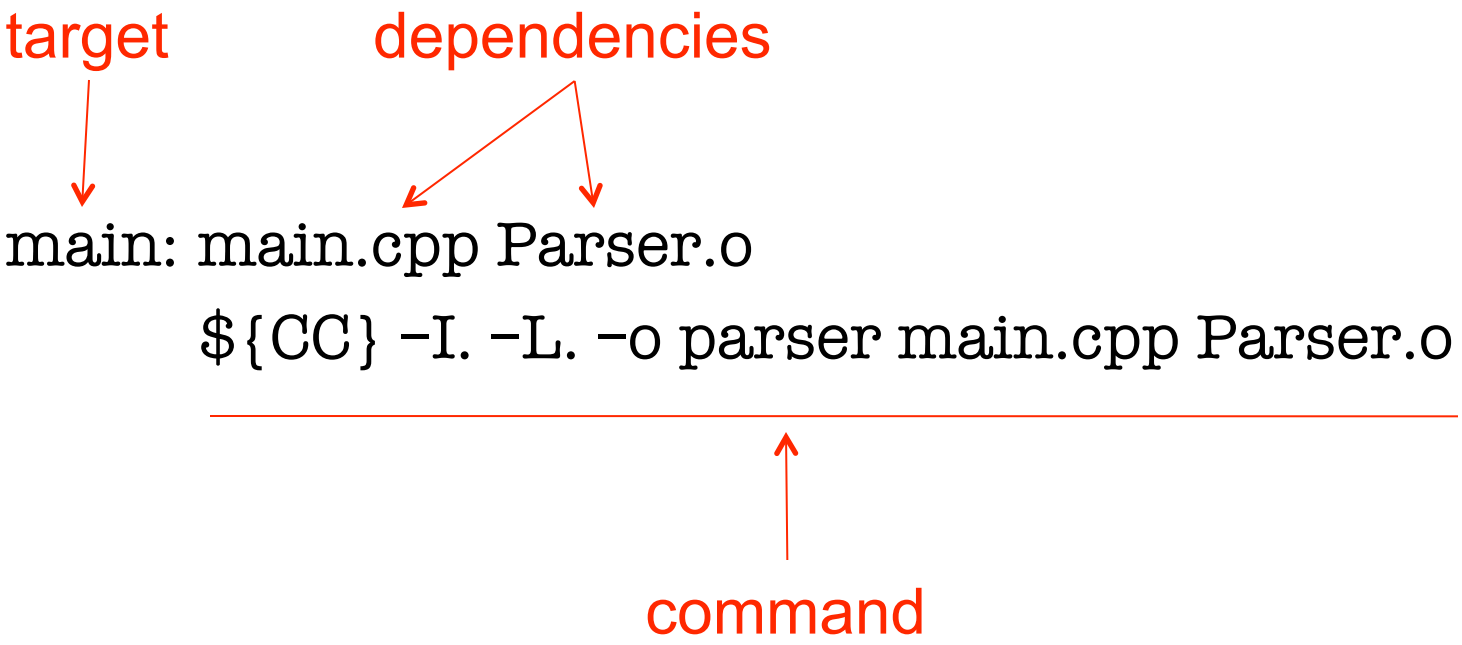
Makefiles

```
CC=g++

all:    main

main: main.cpp Parser.o
        ${CC} -I. -L. -o parser main.cpp Parser.o

clean:
        rm -f parser *.o *~
```

Makefiles

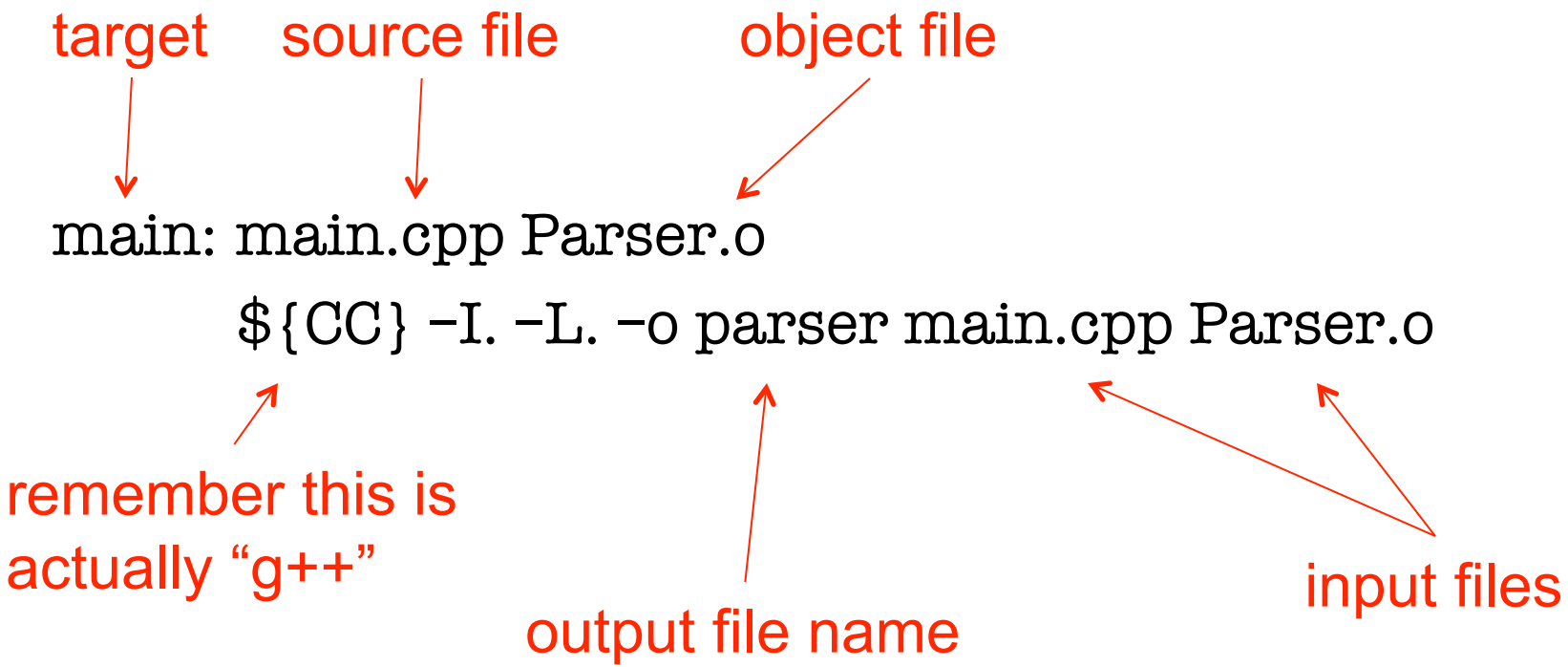
```
CC=g++

all:    main

main: main.cpp Parser.o
        ${CC} -I. -L. -o parser main.cpp Parser.o

clean:
        rm -f parser *.o *~
```

Makefiles: Example Target



To run this target: `make main`

Makefiles: Example Target



Makefiles: Example Target

```
main: main.cpp Parser.o
        ${CC} -I. -L. -o parser main.cpp Parser.o
```

what about these?

-I<path> adds a path for the compiler to search for header files

-L<path> adds a path for the *linker* to search for object files and libraries

Makefiles

- That was one example
- Makefiles are a dark art
- The first 10% will get you 80% of the way
- The rest is usually more than you need, but makes you an expert

Basics : primitive data types

- char (represents byte)
- int
- long
- float
- double
- bool (masks of 0 and 1)
- C-string
- C++ string

Basics : arrays

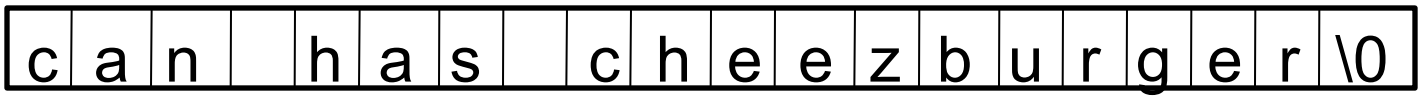
- Contiguous block of memory of `n` cells
- Each cell is the size of the provided type:
  - `char[40]` = 40 * 1 byte per char = 40
  - `double[23]` = 23 * 8 bytes per double = 184

- Arrays also double as pointers (later)
- Also can allocate arrays dynamically (later)

Basics: strings

- In C, strings are arrays of characters:

```
char * str = "can has cheezburger";
```

str = 

str[2] = 'n';

termination char

- In C++, you have the string class:

```
#include <string>

string str1 = "koala";
string str2;
str2 = str1;
```

```
str2 += " in a tree";
if (str2 == str1) ...
if (str1 > str2) ...
```

Basics: strings

- C-strings and C++ strings are functionally equivalent

- C++ `string` provides better encapsulation
  - `char str[5] = "abcd"; char c = str[22];`
    is syntactically ok, semantically bad, but it will run (until there's a crash)
  - `string` will automatically resize
  - and more (adv. I/O will show this)

Basics : control flow

```
switch(variable)
  {
      case A:
              do something;
              break;
       default :
   }
```

if(variable == A)
do something

```
if(condition) {
      do something;
} else {
       do something else;
}
```

if-else clause

Basics : iteration

```
for(int i=0; i<10; i++)
{
   do something;
}
```

loop for 10 times

```
while(condition)
{
  do something;
}
```

loop while condition holds

struct

- Composite object
- Only stores internal data, no methods
- Predecessor to classes (can be extended)

```
struct inflatable {
        char name[20];
        float volume;
        double price;
};
```

In general:

```
struct <name> {
   <members>
};
```

Later:

```
        inflatable    balloon;
        inflatable    tire;
```

union

- Fixed-width, multi-role object
- Only stores enough for biggest type

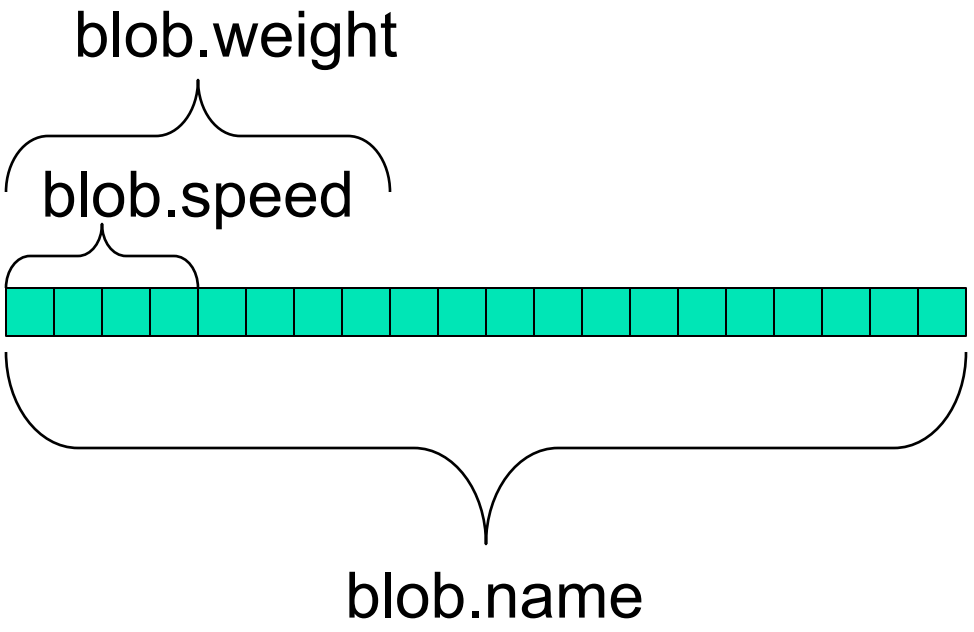
```
struct inflatable {
        char name[20];
        float volume;
        double price;
};
```

```
union blob{
    char name[20];
    float speed;
    double weight;
};
```

inflatable uses (20 + 4 + 8 =) 32 bytes.
blob uses 20 bytes.

union

```
union blob{
    char name[20];
    float speed;
    double weight;
};
```



enum

- Allows for restricted values for a variable.
- Example:

```
enum spectrum { red, orange, yellow, ... };
spectrum signal = yellow;
```

- In reality
  - compiler assigns numbers to each member
  - they only act as masks
  - can assign yourself:

```
enum spectrum { red=1, orange=7, ... };
```

typedef

- Aliases existing types
- General form:
  - typedef <original type> <alias name>
- Example
  - Nodes in a red-black tree are ‘red’ or ‘black’
  - typedef enum { ‘RED’, ‘BLACK’ } Color;
    - original type: enum {‘RED’, ‘BLACK’}
    - alias: Color
  - Later:

    Color c = RED;

Aliased new types

```
typedef struct AbstractDataType {
   struct AbstractDataType *parent;
   void *data;
   std::string description;
} ADT;
```

Aliased new types

```
typedef struct AbstractDataType {
   struct AbstractDataType *parent;
   void *data;
   std::string description;
} ADT;

typedef <type> <name>;
```

Aliased new types

```
typedef struct AbstractDataType {
   struct AbstractDataType *parent;
   void *data;
   std::string description;
} ADT;

typedef <type> <name>;
```

Aliased new types

- Pointers are also fair game:

```
typedef struct AbstractDataType {
   struct AbstractDataType *parent;
   void *data;
   std::string description;
} *ADT;
typedef <type> <name>;
```

In use: ADT ptr = new ADT();

Next lecture

Part 1 of the C++ Standard Library:

The C Library