C++:

- an object-oriented language like Objective-C, but stricter, in terms of:
  - data typing (strongly-typed lang.)
    ⇒ discourages (void *) castings
  - data access/promiscuity
    ⇒ objects specify what others can read/write

- basic structure of object similar to obj-C, an object (called class) still has data & methods

data.h:

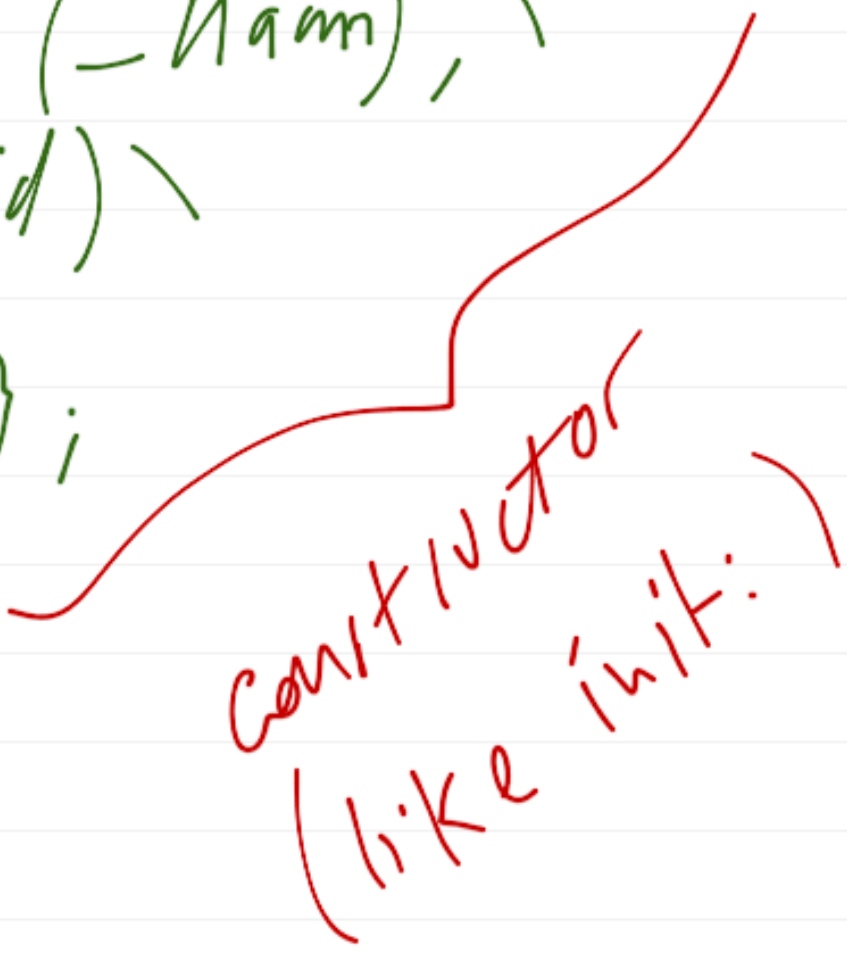
```
#ifndef DATA_H
#define DATA_H
⋮
```

allows multiple #include "data.h"

#endif

(#import did this for you)

```
class data_t
{
  public:
  data_t(std::string _nam="",
         int _id = -1) : \
    name(_nam), \
    id(_id) \
  { } ;
```



```
// destructor - how to delete
   (free or release) object
   -- if no "new" (malloc)
      data elements,
      can use default dest.
// friends (output operator)
friend std::ostream&
    operator<<(std::ostream& s,
  const data_t& rhs)
{ s << rhs.name << " ";
  s << rhs.id << std::endl;
  return s;
}
```

```
friend std::ostream&
    operator<<(std::ostream& s,
               data_t *rhs)
{
    return (s << (*rhs));
}
private:
  std::string name;
  int         id;
};//end of class
```

data members

- What are all those << ?
- stream operators
- instead of stdout, we have
  std::cout
- instead of stderr, we have
  std::cerr
- basic program:

```
#include <iostream>
int main(int argc, char *argv[])
{
  std::cout << "hello world";
}
```

- introduce data object

```
#include <iostream>
int main()
{
    data_t data("Bob", 123);
    std::cout << data << std::endl;
}
```

object writes itself

- in reality, above gets translated to:

```
std::cout.operator<<(const data_t &)
```

function call

std::cout.operator<<(const data_t &)

- things to note about this:
- function belongs to cout object
- normally cout "knows" about int, double, float, etc, but not some user-defined type like data_t
- above overloads operator<<

- for cout to be able to print contents of data_t it needs to access its data members

- but these are private: only data_t can access private data_t members UNLESS object is a friend

- so the lines

```
s << rhs.name;
s << rhs.id;
```

get translated to

```
s.operator<<(std::string)
s.operator<<(int)
```

- which are basic types
- std::string is a string object, not unlike NSString

- What about constructor?

```
data_t(std::string _name = "",
       int _id = -1) : \
  name(_name), \
  id(_id) \
{ , };
```

arguments

Constructor

init code, if any

- the = "" and = -1
  are the default values
- caller can use:

```
data_t d1("Bob", 123);
data_t d2();
```

default values "", -1 used

- What does `const data_t &` mean?

`const` says you promise not to alter (assign) this object (read-only)

`&` means pass-by-reference (more efficient than pass-by-value, but distinct from pass as pointer)

- suppose we had declared argument simply as data_t then what?

pass-by-value, which copies all of data_t via copy constructor

```
data_t(const data_t& rhs)
{
    name = rhs.name;
    id = rhs.id;
}
```

assignment operator, data_t needs one too

```
data_t& operator=(const data_t& rhs)
{
    if(this != &rhs){
        name = rhs.name;
        id = rhs.id;
    }
    return *this;
}
```

- this is a reserved keyword for a pointer to this object
- the "big three":
  - copy constructor
  - assignment operator
  - destructor
- every object (class) should provide these

- what about inheritance?

```
class object_t
{ public:
    //constructors
    //destructors
    //operators
    //member functions
  private:
    //...
};
class sphere_t : public object_t
{
  :
}
```

inherit from parent

- virtual (polymorphic) functions

```
class object_t
{
  :
  virtual double hits(const
                      vec_t &,
    :
                      const
                      vec_t &);
};

class sphere_t : public object_t
{
  double hits(const vec_t &,
              const vec_t &);
};
```

- How to compile a c++ program:

- Suppose we have:

data.h – interface:

data.cpp

main.cpp

class data_t
{
:
:
};

implementation (can just be:)

#include <iostream>
#include "data.h"

Contains main()

- Compile with these commands:

```
g++ -c -I. -g main.cpp
g++ -c -I. -g data.cpp
```

- there are the two compile stages, producing

```
main.o
data.o
```

- then link:

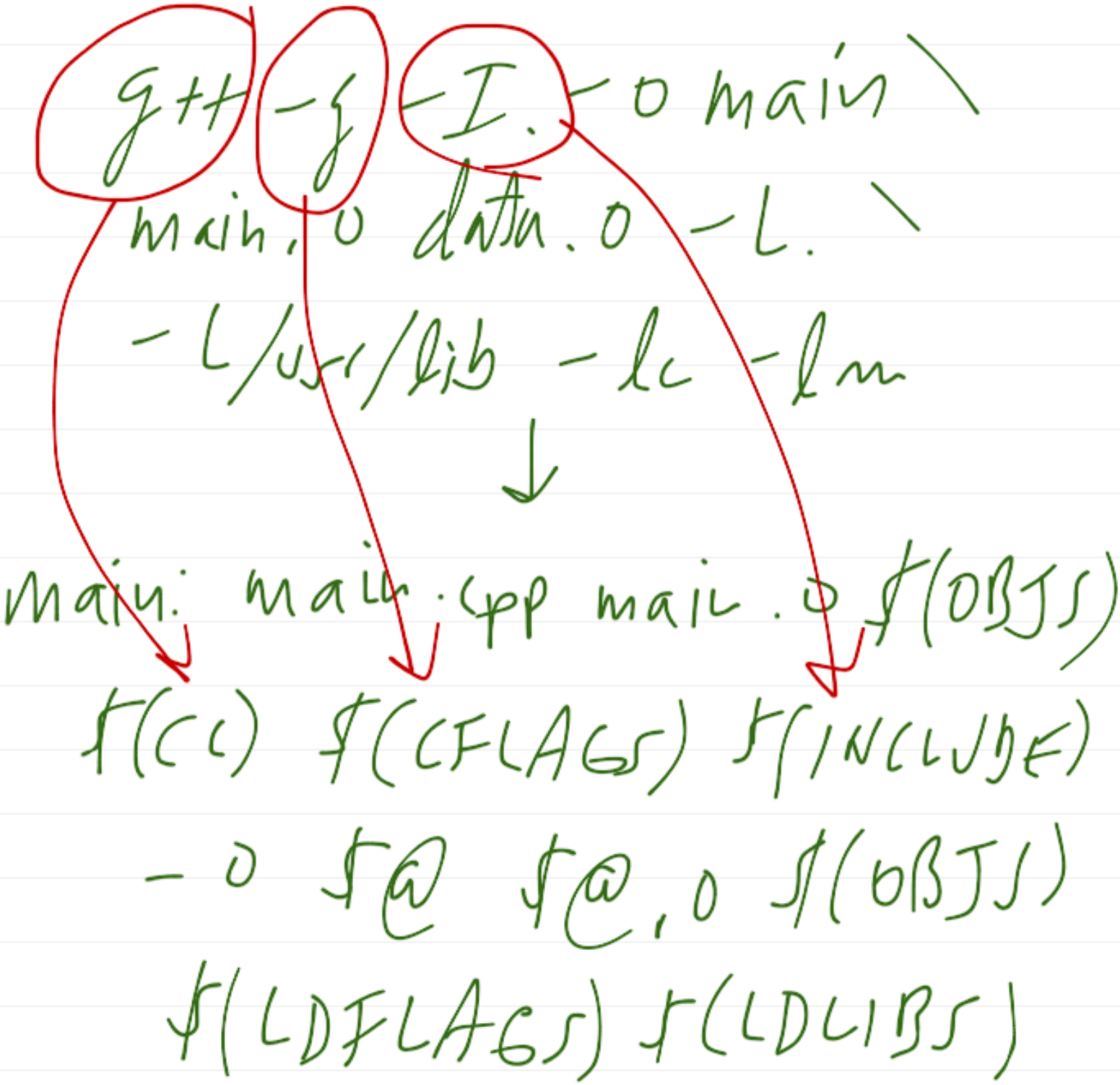
```
g++ -g -I. -o main \
main.o data.o -L. \
-L/usr/lib -lc -lm
```

- let's walk backwards:

```
g++ -g -I. -o main \
main.o data.o -L. \
-L/usr/lib -lc -lm
```

↓

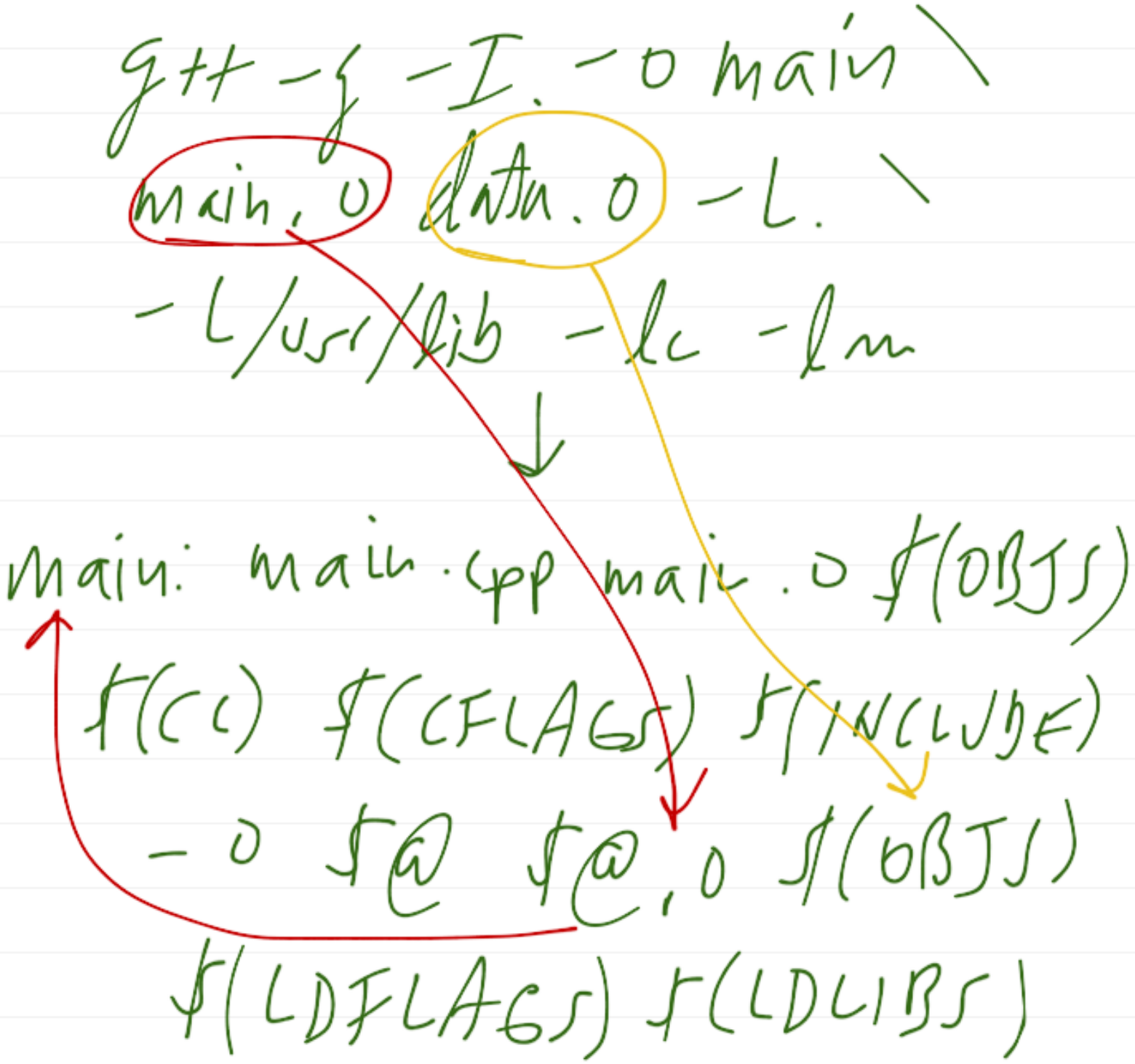
```
main: main.cpp main.o $(OBJS)
	$(CC) $(CFLAGS) $(INCLUDE)
	-o $@ $@.o $(OBJS)
	$(LDFLAGS) $(LDLIBS)
```

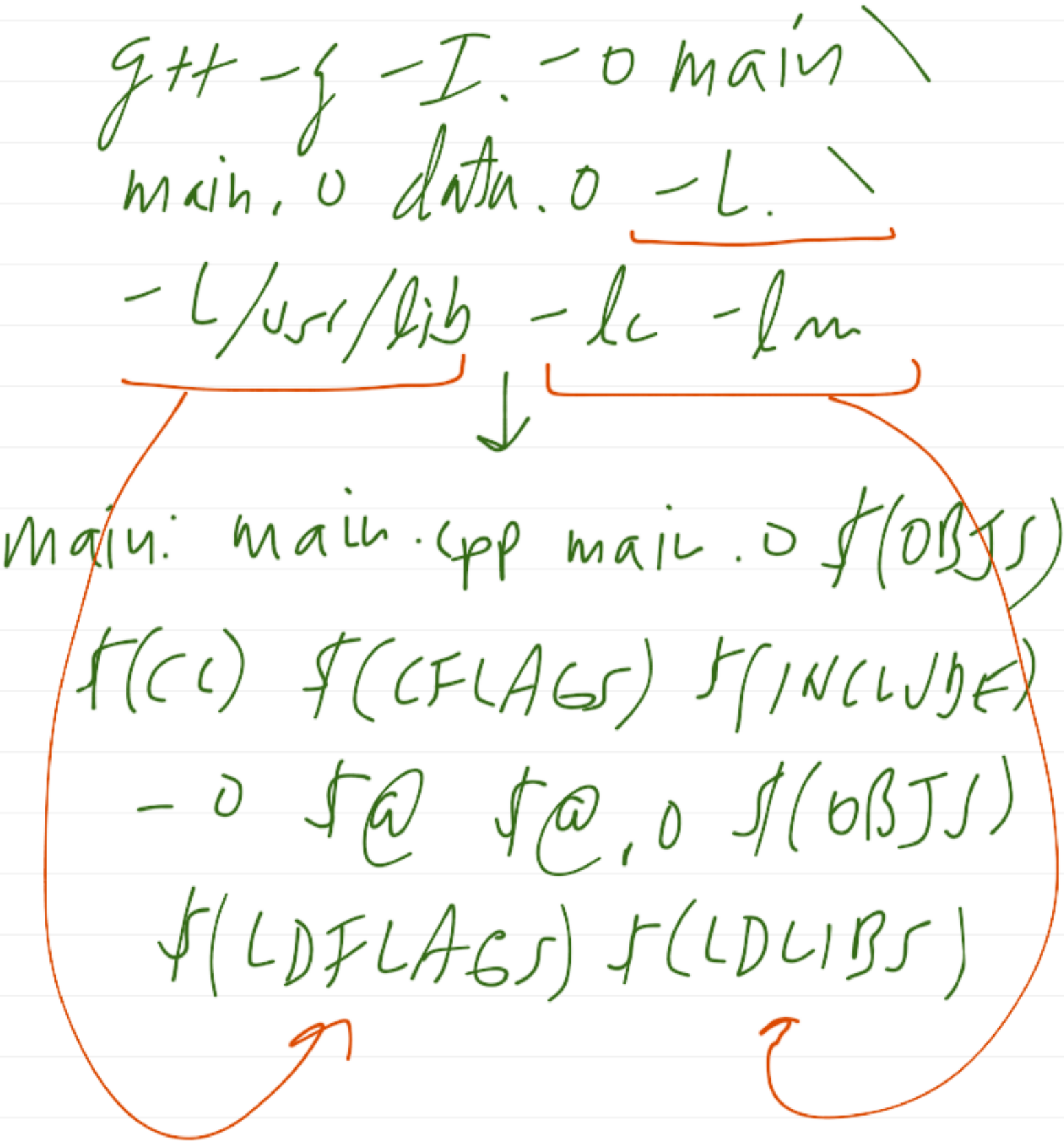

g++ -g -I. -o main \
main.o data.o -L. \
-L/usr/lib -lc -lm
main: main.cpp main.o $(OBJS)
$(CC) $(CFLAGS) $(INCLUDE)
-o $@ $@.o $(OBJS)
$(LDFLAGS) $(LDLIBS)

g++ -g -I. -o main \
main.o data.o -L. \
-L/usr/lib -lc -lm

↓

main: main.cpp main.o $(OBJS)
$(CC) $(CFLAGS) $(INCLUDE)
-o $@ $@.o $(OBJS)
$(LDFLAGS) $(LDLIBS)


g++ -g -I. -o main \
main.o data.o -L. \
-L/usr/lib -lc -lm
main: main.cpp main.o $(OBJS)
$(CC) $(CFLAGS) $(INCLUDE)
-o $@ $@.o $(OBJS)
$(LDFLAGS) $(LDLIBS)


g++ -g -I. -o main \
main.o data.o -L. \
-L/usr/lib -lc -lm
main: main.cpp main.o $(OBJS)
$(CC) $(CFLAGS) $(INCLUDE)
-o $@ $@.o $(OBJS)
$(LDFLAGS) $(LDLIBS)

main: main.cpp main.o $(OBJS)

$(CC) $(CFLAGS) $(INCLUDE)

-o $@ $@.o $(OBJS)

$(LDFLAGS) $(LDLIBS)

- déjà deux variables:

CC = g++

INCLUDE = -I.

CFLAGS = -g

LDFLAGS = -L. -L/usr/lib

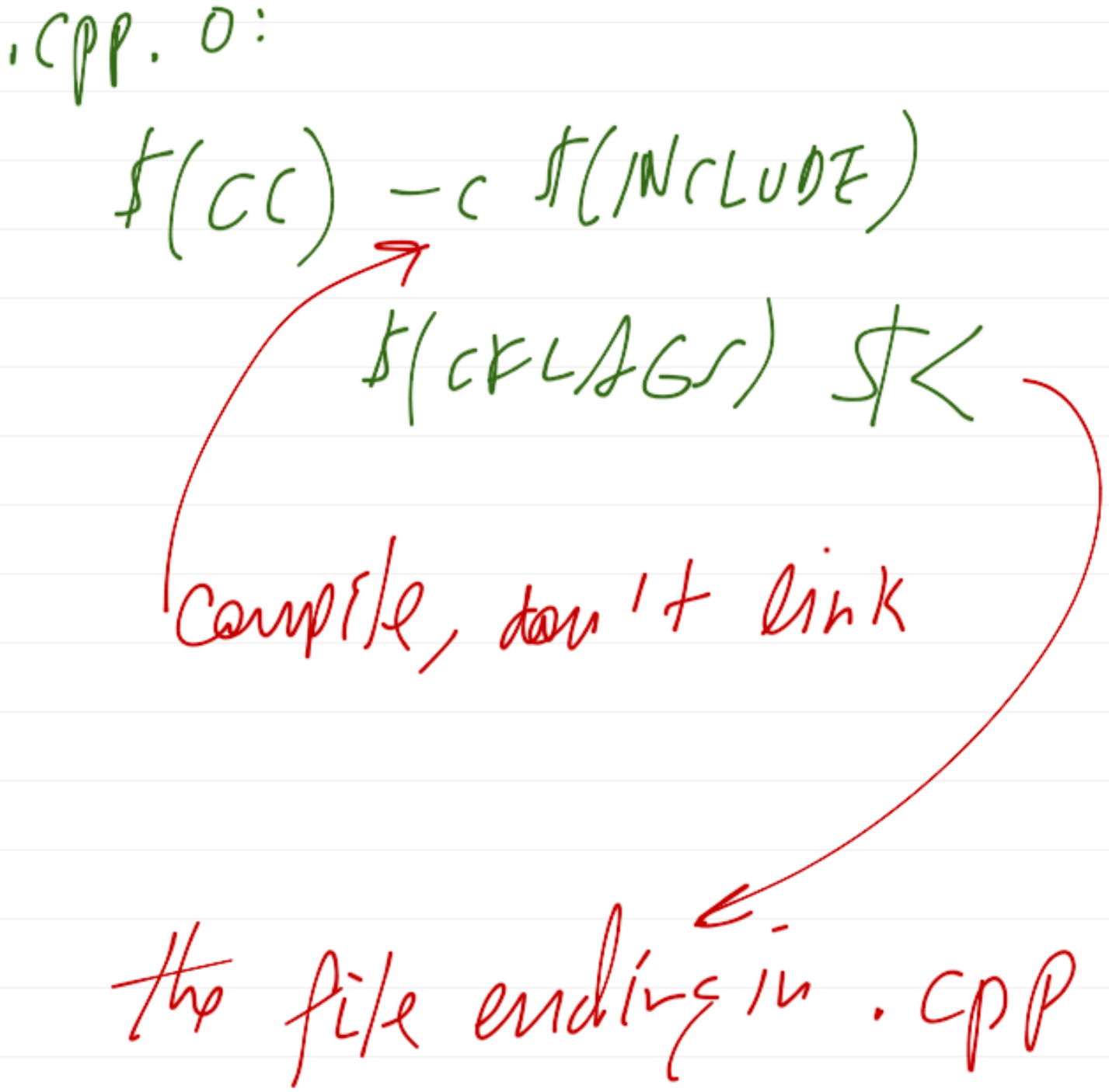
OBJS = data.o

- The above with the default target all: specified:

all: main

defines the link stage recursively

- All that's missing is to tell the Makefile how to compile (without linking) .c files


.cpp.o:
$(CC) -c $(INCLUDE)
$(CFLAGS) $<
compile, don't link
the file ending in .cpp

- this sets up both link and compile stages recursively via these targets:

```
all:

main:

.cpp.o:
```

- add one more to clean up:

```
clean:
    rm -f *.o core main
```