C Compilation Process

Lecture 6

Christian A. Pagot

Universidade Federal da Paraíba
Centro de Informática

About this Presentation

- This presentation is based, under permission, on the excellent, and very didactic, article entitled "**The C++ Compilation Model**", written by **David Röthlisberger**.
- Students are encouraged to read the original article, which is available at:
  - http://david.rothlis.net/c/compilation_model

C Program

- A C program consists of one or more **source files**.
- Each file usually contain:
  - C code.
  - Preprocessor directives (**`#define`**, **`#include`**, **`#ifdef`**, etc.).

Building a C Program with GCC

Program composed by one C source file

sum.c

```c
int x[5] = { 1, 2, 3, 4, 5 };

int Sum( void ) {
    int i;
    int acc = 0;

    for ( i = 0; i < 5; i++ )
        acc += x[i];

    return acc;
}

int main( void ) {
    int sum_value = Sum();

    return sum_value;
}
```

All warnings are raised, and they will be reported as errors!

```
~$ gcc -Wall -Werror sum.c -o sum
```

...and *voilà*!!!
We've got an **executable file** on disc!

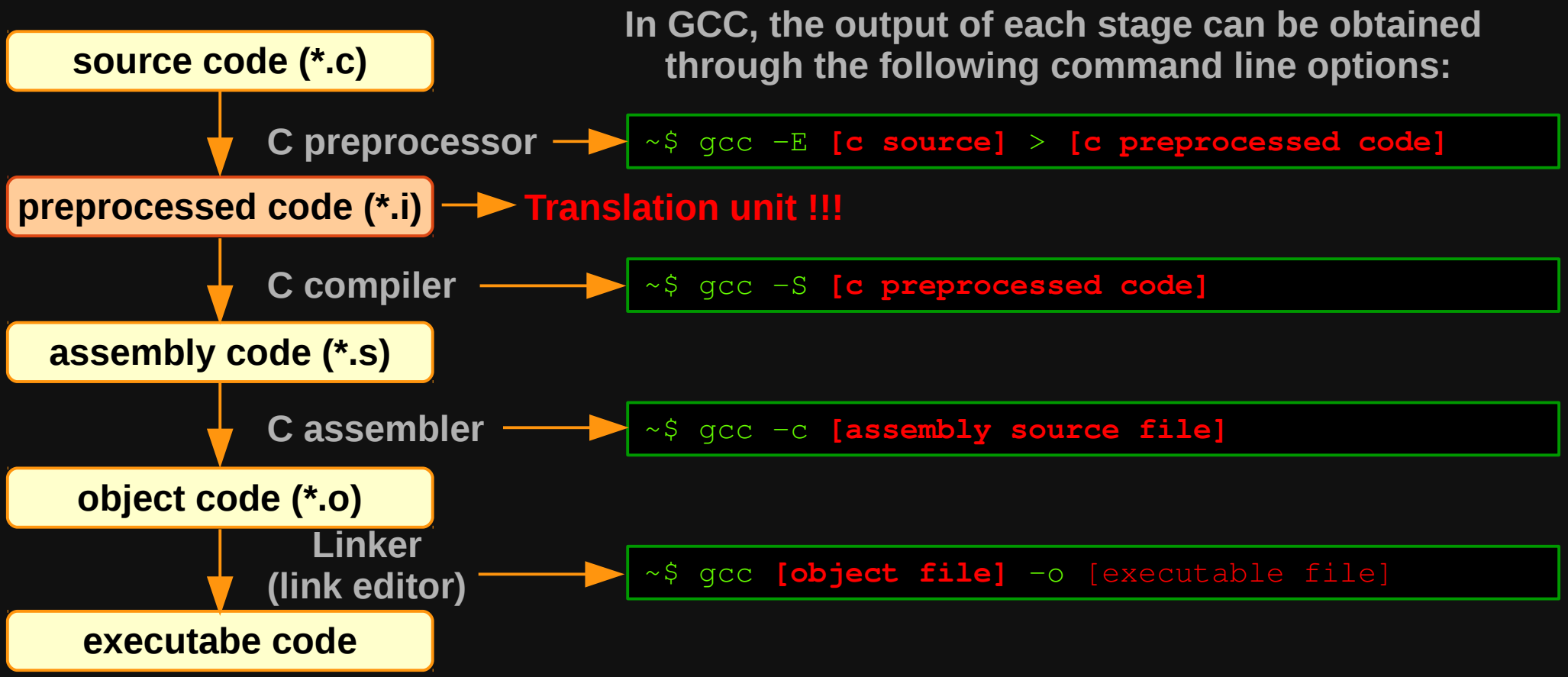
```
~$ ./sum; echo $?
```

However, what exactly **happened** during the "**compilation**" process?

DIY!

Step-by-step Building

Building an **executable** from source actually involves **several steps**.

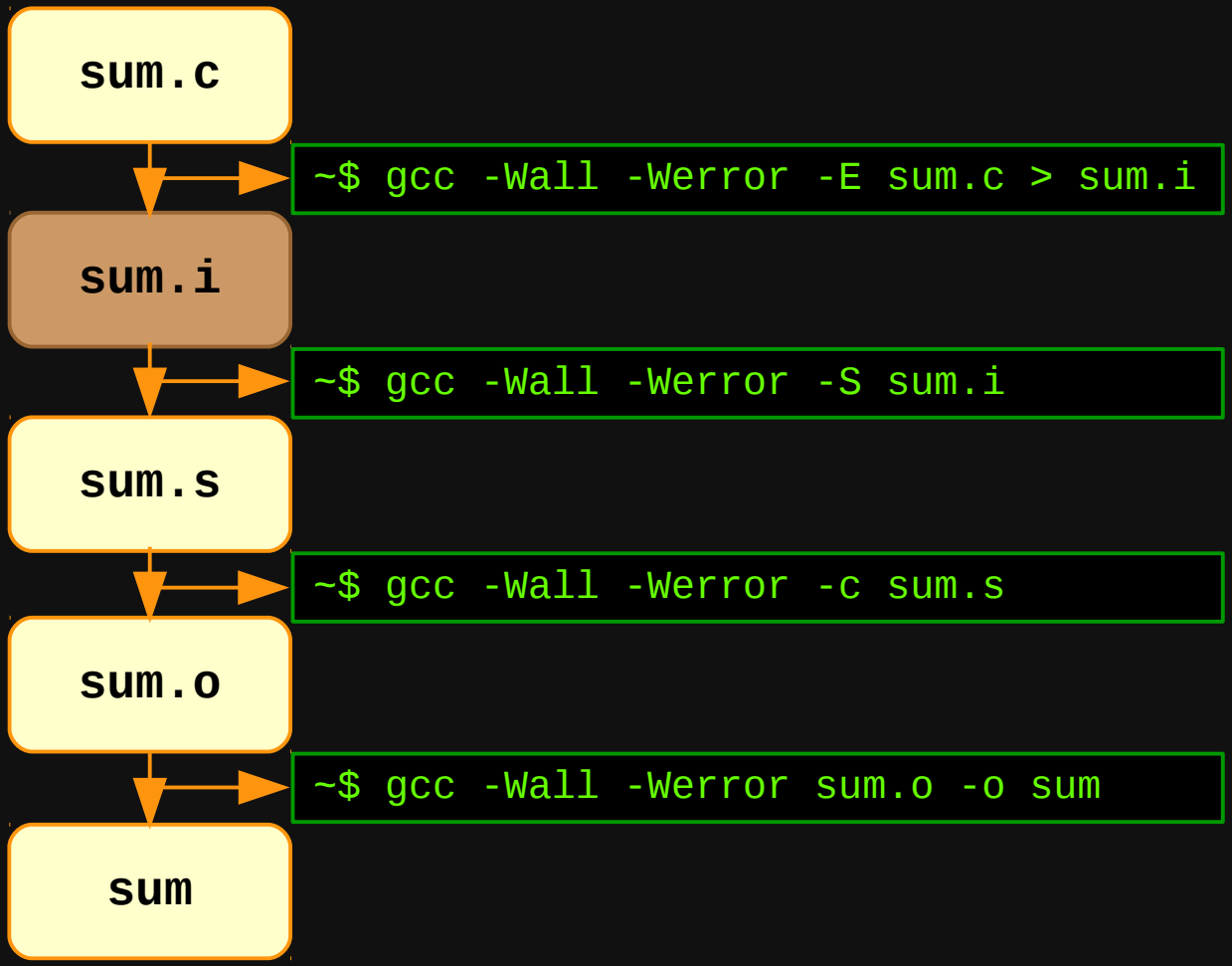


In GCC, the output of each stage can be obtained through the following command line options:

Step-by-step Building

sum.c

```c
int x[5] = { 1, 2, 3, 4, 5 };

int Sum( void ) {
    int i;
    int acc = 0;

    for ( i = 0; i < 5; i++ )
        acc += x[i];

    return acc;
}

int main( void ) {
    int sum_value = Sum();

    return sum_value;
}
```



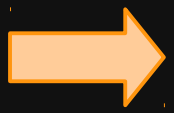
Program Composed of +1 C Source

Splitting the previous code into two separate C source files

sum.c

```c
int x[5] = { 1, 2, 3, 4, 5 };

int Sum( void ) {
    int i;
    int acc = 0;

    for ( i = 0; i < 5; i++ )
        acc += x[i];

    return acc;
}
```

```c
int main( void ) {
    int sum_value = Sum();

    return sum_value;
}
```



sumdata.c

```c
int x[5] = { 1, 2, 3, 4, 5 };

int Sum( void ) {
    int i;
    int acc = 0;

    for ( i = 0; i < 5; i++ )
        acc += x[i];

    return acc;
}
```

summain.c

```c
int main( void ) {
    int sum_value = Sum();

    return sum_value;
}
```

DIY!

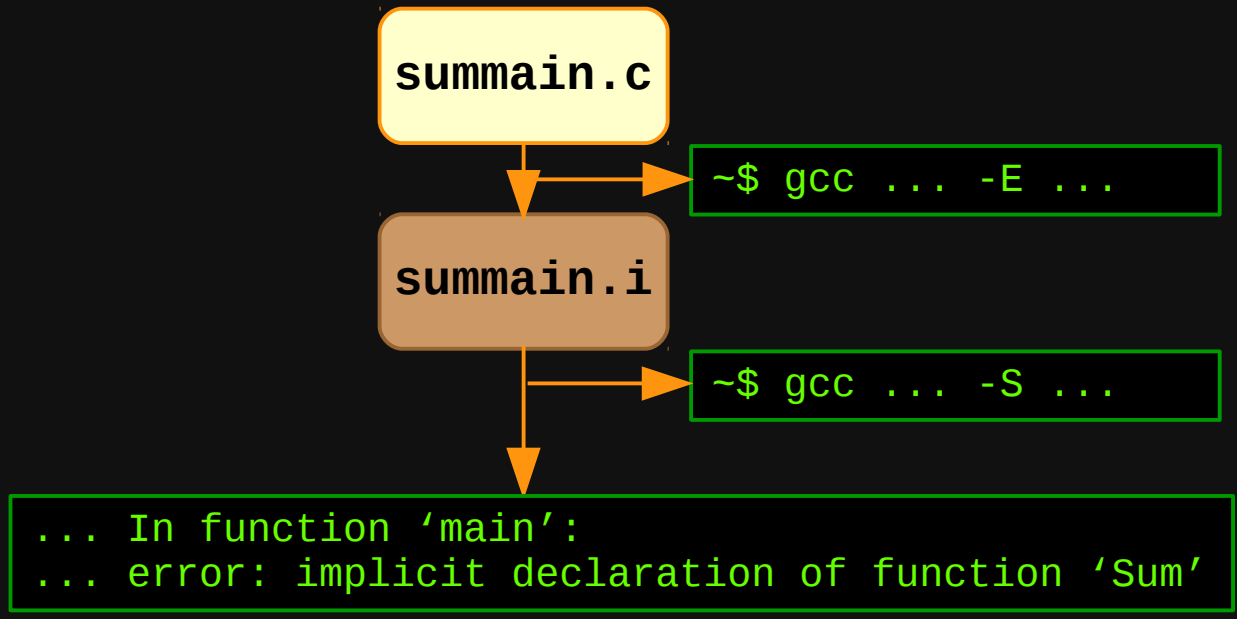
Building a Two C Source File Program

sumdata.c

```c
int x[5] = { 1, 2, 3, 4, 5 };

int Sum( void ) {
    int i;
    int acc = 0;

    for ( i = 0; i < 5; i++ )
        acc += x[i];

    return acc;
}
```

summain.c

```c
int main( void ) {
    int sum_value = Sum();

    return sum_value;
}
```



The **compiler** has no idea about the function **Sum()**!

Thus, we will have to **declare** the **Sum()** function within **summain.c**! A declaration informs that **Sum()** is (hopefully) **defined** somewhere else!

Building a Two Source File C Prog.

sumdata.c

```c
int x[5] = { 1, 2, 3, 4, 5 };

int Sum( void ) {
    int i;
    int acc = 0;

    for ( i = 0; i < 5; i++ )
        acc += x[i];

    return acc;
}
```

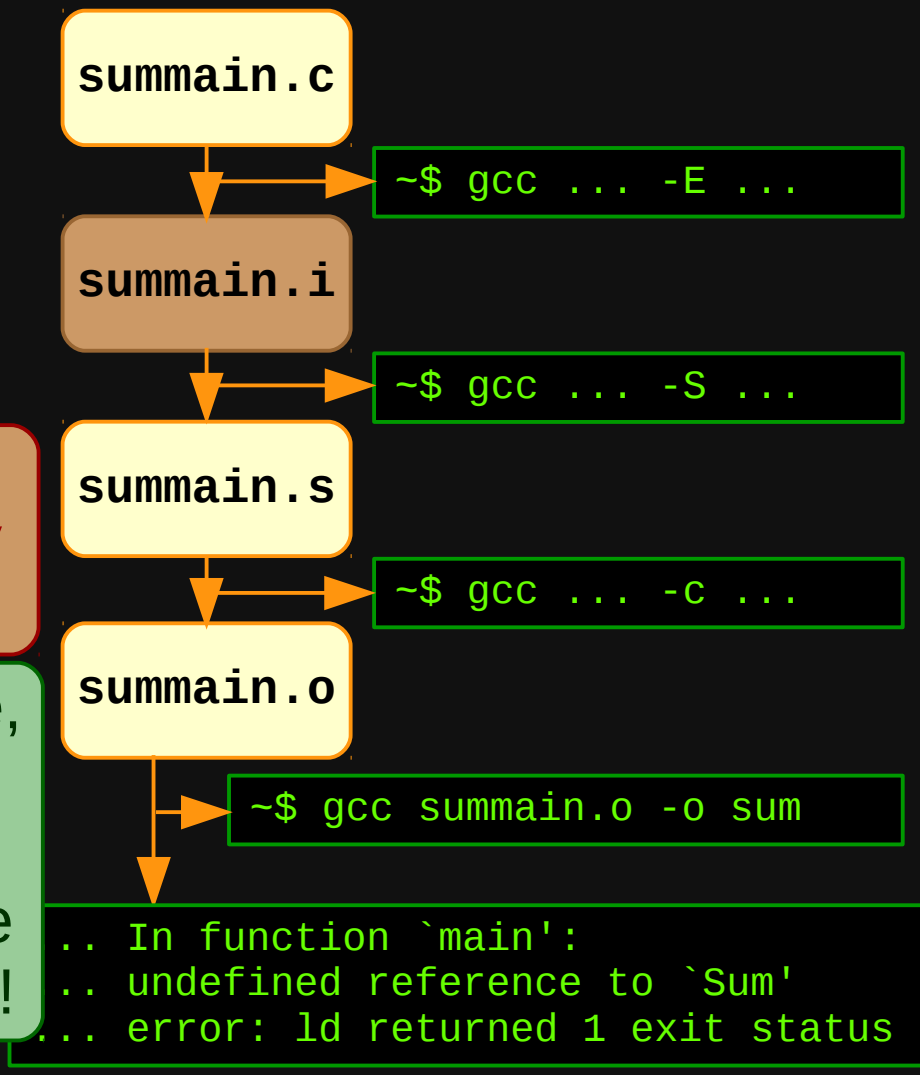
summain.c

Sum() declaration

```c
int Sum( void );

int main( void ) {
    int sum_value = Sum();

    return sum_value;
}
```

The **linker** could not find the **entry point** of **Sum()**!

We must provide, to the linker, the object code generated by the **Sum()** definition!



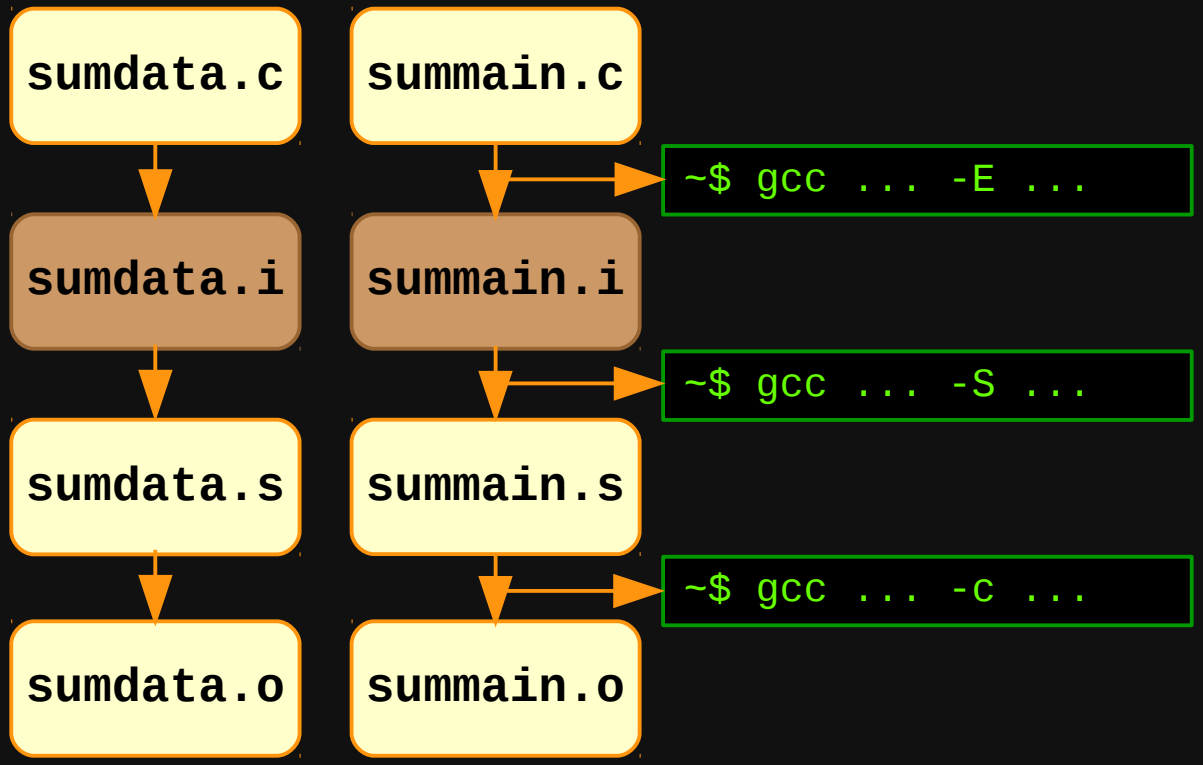
Building a Two Source File C Prog.

sumdata.c

```c
int x[5] = { 1, 2, 3, 4, 5 };

int Sum( void ) {
    int i;
    int acc = 0;

    for ( i = 0; i < 5; i++ )
        acc += x[i];

    return acc;
}
```

summain.c

```c
int Sum( void );

int main( void ) {
    int sum_value = Sum();

    return sum_value;
}
```



Object File Symbols

Assembly code generated by **summain.c**.

summain.c

```c
int Sum( void );

int main( void ) {
    int sum_value = Sum();

    return sum_value;
}
```

summain.s

```
        .file   "summain.c"
        .text
        .globl  main
        .type   main, @function
main:
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $16, %rsp
        call    Sum
        movl    %eax, -4(%rbp)
        movl    -4(%rbp), %eax
        leave
        ret
```

```
~$ man nm
```

```
~$ nm summain.o
0000000000000000 T main
                 U Sum
```

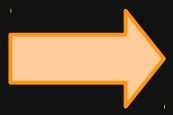
T : symbol is in the `.text` (code) section.
U : symbol is undefined.

Object File Symbols

Assembly code generated by **sumdata.c**.

sumdata.c

```c
int x[5] = { 1, 2, 3, 4, 5 };

int Sum( void ) {
    int i;
    int acc = 0;

    for ( i = 0; i < 5; i++ )
        acc += x[i];

    return acc;
}
```



sumdata.s

```
        .file   "sumdata.c"
        .globl  x
        .data
        .align 16
        .type   x, @object
        .size   x, 20
x:
        .long   1
            . . .
        .long   5
        .text
        .globl  Sum
        .type   Sum, @function
Sum:
        pushq   %rbp
        movq    %rsp, %rbp
            . . .
        movl    -4(%rbp), %eax
        popq    %rbp
        ret
```

```
~$ nm sumdata.o
0000000000000000 T Sum
0000000000000000 D x
```

T : symbol is in the `.text` (code) section.
D : symbol is in `.data` section..

We might also inspect object files with **objdump**!

Object File Symbols

nm vs. objdump

```
~$ nm sumdata.o
0000000000000000 T Sum
0000000000000000 D x
```

T : symbol is in the `.text` (code) section.
D : symbol is in `.data` section..

```
~$ objdump -t sumdata.o

sumdata.o:     file format elf64-x86-64

SYMBOL TABLE:
0000000000000000 l    df *ABS*  0000000000000000 sumdata.c
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l    d  .note.GNU-stack    0000000000000000 .note.GNU-stack
0000000000000000 l    d  .comment   0000000000000000 .comment
0000000000000000 g     O .data  0000000000000014 x
0000000000000000 g     F .text  0000000000000032 Sum
```

l : local symbol
g : global symbol
d: debug symbol

f : file
O : object
F: function

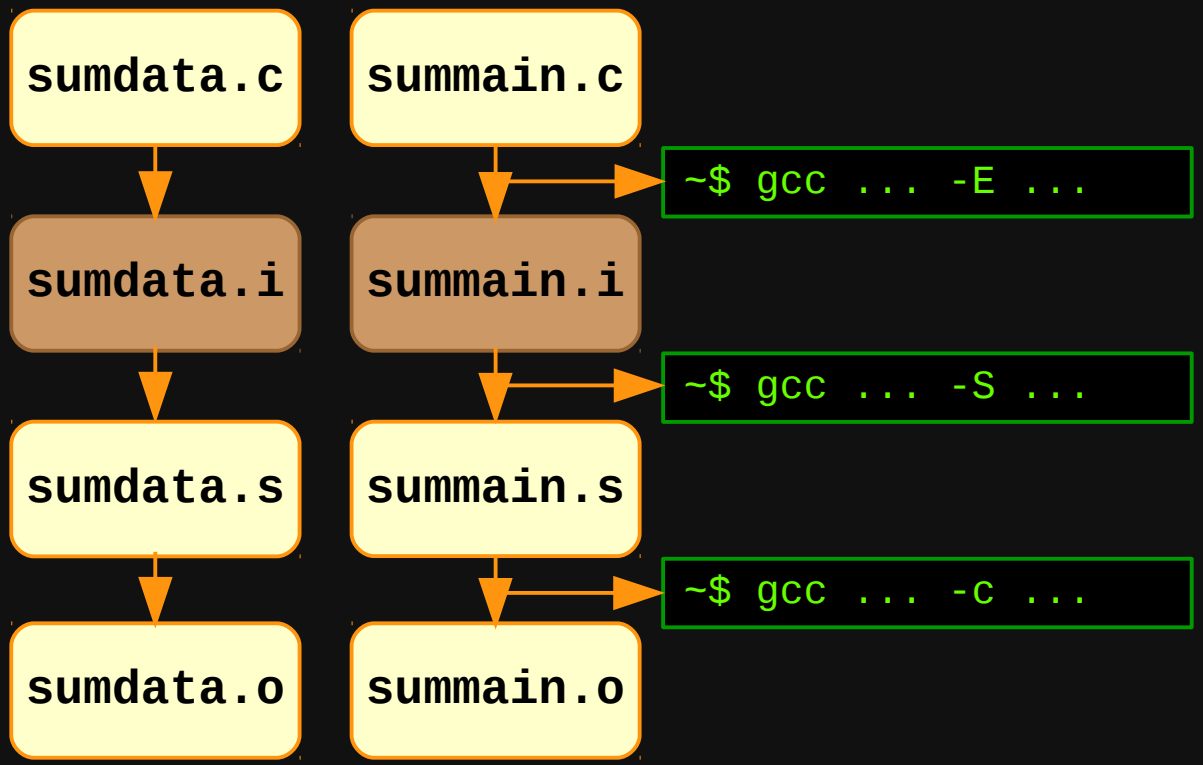
Building a Two Source File C Prog.

sumdata.c

```c
int x[5] = { 1, 2, 3, 4, 5 };

int Sum( void ) {
    int i;
    int acc = 0;

    for ( i = 0; i < 5; i++ )
        acc += x[i];

    return acc;
}
```

summain.c

```c
int Sum( void );

int main( void ) {
    int sum_value = Sum();

    return sum_value;
}
```




```
~$ nm summain.o
0000000000000000 T main
                 U Sum
```

```
~$ nm sumdata.o
0000000000000000 T Sum
0000000000000000 D x
```

The linker now knows where the definition of Sum( ) is!

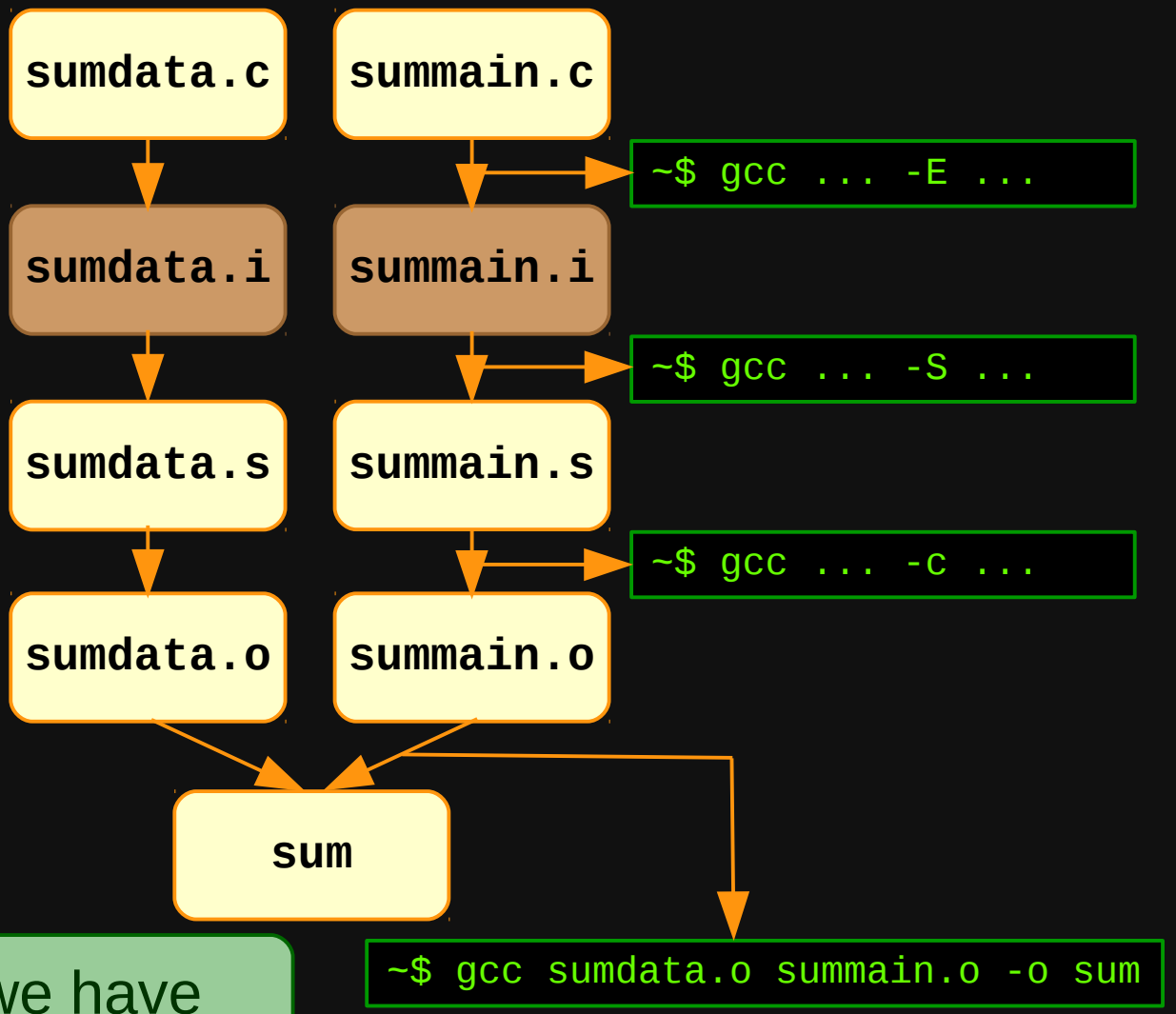
Building a Two Source File C Prog.

sumdata.c

```c
int x[5] = { 1, 2, 3, 4, 5 };

int Sum( void ) {
    int i;
    int acc = 0;

    for ( i = 0; i < 5; i++ )
        acc += x[i];

    return acc;
}
```

summain.c

```c
int Sum( void );

int main( void ) {
    int sum_value = Sum();

    return sum_value;
}
```



Now we have an **executable**!

Building a Two Source File C Prog.

sumdata.c

```
int x[5] = { 1, 2, 3, 4, 5 };

int Sum( void ) {
    int i;
    int acc = 0;

    for ( i = 0; i < 5; i++ )
        acc += x[i];

    return acc;
}
```

sumdata.s

```
            . . .
        .long   5
        .text
        .globl  Sum
        .type   Sum, @function

Sum:
        pushq   %rbp
        movq    %rsp, %rbp
            . . .
        movl    -4(%rbp), %ea
        popq    %rbp
        ret
```

summain.c

```
int Sum( void );

int main( void ) {
    int sum_value = Sum();

    return sum_value;
}
```

summain.s

```
        .file   "summain.c"
        .text
        .globl  main
        .type   main, @functi
main:
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $16, %rsp
        call    Sum
        movl    %eax, -4(%rbp)
        movl    -4(%rbp), %eax
        leave
        ret
```

```
~$ objdump -td sum
```

```
  . . .
00000000004004ed <main>:
  4004ed:push   %rbp
  4004ee:mov    %rsp,%rbp
  4004f1:sub    $0x10,%rsp
  4004f5:callq  400502 <Sum>
  . . .
0000000000400502 <Sum>:
  400502:push   %rbp
  400503:mov    %rsp,%rbp
  400506:movl   $0x0,-0x4(%rbp)
  40050d:movl   $0x0,-0x8(%rbp)
  400514:jmp    400529 <Sum+0x27>
  400516:mov    -0x8(%rbp),%eax
  . . .
```

The **linker** substitute symbols for **real addresses**!

Object Declaration and Definition

- Object Definition
  - Must occur only in one place.
  - Specifies the object type and makes the compiler reserve memory space.
- Object Declaration
  - May occur in more than one place.
  - Serves as a description of an object that is defined somewhere else.

Object Declaration and Definition

Example (back to our C program):

summain.c

```c
int Sum( void );

int main( void ) {
    int sum_value = Sum();

    return sum_value;
}
```

sumdata.c

```c
int x[5] = { 1, 2, 3, 4, 5 };

int Sum( void ) {
    int i;
    int acc = 0;

    for ( i = 0; i < 5; i++ )
        acc += x[i];

    return acc;
}
```

Declaration: does not reserve space. It is just a hint for the compiler.

Declaration: does not reserve space at compile time.

Definition: store the object in `.data` section.

Definition: reserve space in `.text` section.

summain.s

```
        .file   "summain.c"
        .text
        .globl  main
        .type   main, @function
main:
        . . .
        call    Sum
        movl    %eax, -4(%rbp)
        . . .
```

sumdata.s

```
        .file   "sumdata.c"
        .globl  x
        .data
        .align 16
        .type   x, @object
        .size   x, 20
x:
        .long   1
          . . .
        .long   5
        .text
        .globl  Sum
        .type   Sum, @function
Sum:
        pushq   %rbp
        movq    %rsp, %rbp
          . . .
        movl    -4(%rbp), %eax
        popq    %rbp
        ret
```

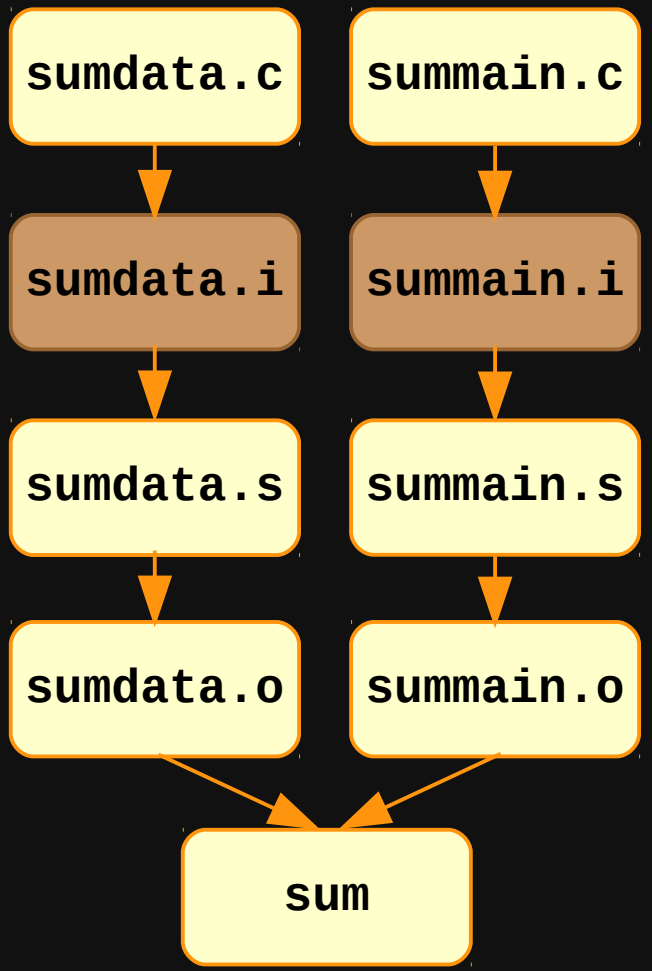
Hey! We are typing **too much**!

To build the sum program (according to the diagram to the left) we have to type the following commands on the prompt:

```
~$ gcc -Wall -Werror -E summain.c > summain.i
~$ gcc -Wall -Werror -S summain.i
~$ gcc -Wall -Werror -c summain.s
~$ gcc -Wall -Werror -E sumdata.c > sumdata.i
~$ gcc -Wall -Werror -S sumdata.i
~$ gcc -Wall -Werror -c sumdata.s
~$ gcc -Wall -Werror summain.o sumdata -o sum
```

How we can automate that?

Yep! Make !!!



Make

GNU Make is a tool which controls the generation of executables and other non-source files of a program from the program's source files.

GNU.org

Make

- Capabilities
  - Enables the end user to build and install programs.

    Details of the operations are recorded in the **Makefile**.
  - Figures out automatically which files it needs to update.
  - Is not limited to any particular language.
  - Control installing or deinstalling a package.

Rules and Targets

- A **rule** indicates **how to build** the **target** file from **source** files.
- General format:

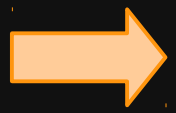
Makefile

```
target: dependencies...
    commands
    ...
```

“Making” a Simple C Program

assembly1.c

```
int x = 10;
int y;

main() {
    int w;
    y = 20;
    w = x + y;
    return 0;
}
```



It could actually be built with just one command:

```
~$ gcc -Wall -Werror assembly1.c -o assembly1
```

However, we still want to generate all intermediary files, in order to be able to inspect them after the building process. In this case, the commands to be issued at the prompt should be:

```
~$ gcc -Wall -Werror -E assembly1.c > assembly1.i
~$ gcc -Wall -Werror -S assembly1.i
~$ gcc -Wall -Werror -c assembly1.s
~$ gcc -Wall -Werror assembly1.o -o assembly1
```

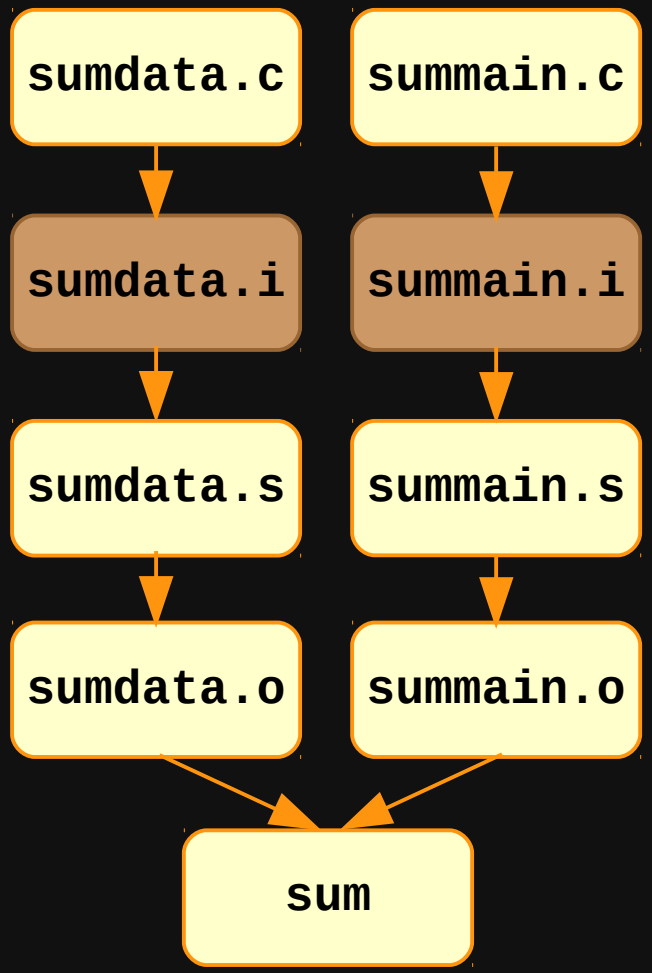
One possible Makefile:

```
assembly1: assembly.c
    gcc -Wall -Werror -E assembly1.c > assembly1.i
    gcc -Wall -Werror -S assembly1.i
    gcc -Wall -Werror -c assembly1.s
    gcc -Wall -Werror assembly1.o -o assembly1
```

At the prompt:

```
~$ make assembly1
```

Now, let's “make” our last program...



Makefile

```
sumdata.o: sumdata.c
    gcc -Wall -Werror -E sumdata.c > sumdata.i
    gcc -Wall -Werror -S sumdata.i
    gcc -Wall -Werror -c sumdata.s

summain.o: summain.c
    gcc -Wall -Werror -E summain.c > summain.i
    gcc -Wall -Werror -S summain.i
    gcc -Wall -Werror -c summain.s

sum: sumdata.o summain.o
    gcc -Wall -Werror summain.o sumdata.o -o sum
```

Removing Intermediary&Exec Files

Makefile

```
sumdata.o: sumdata.c
    gcc -Wall -Werror -E sumdata.c > sumdata.i
    gcc -Wall -Werror -S sumdata.i
    gcc -Wall -Werror -c sumdata.s

summain.o: summain.c
    gcc -Wall -Werror -E summain.c > summain.i
    gcc -Wall -Werror -S summain.i
    gcc -Wall -Werror -c summain.s

sum: sumdata.o summain.o
    gcc -Wall -Werror summain.o sumdata.o -o sum

.PHONY: clean

clean:
    rm *.i *.s *.o sum
```

Enough on **Make** for now!
Back to the **C compilation model**...

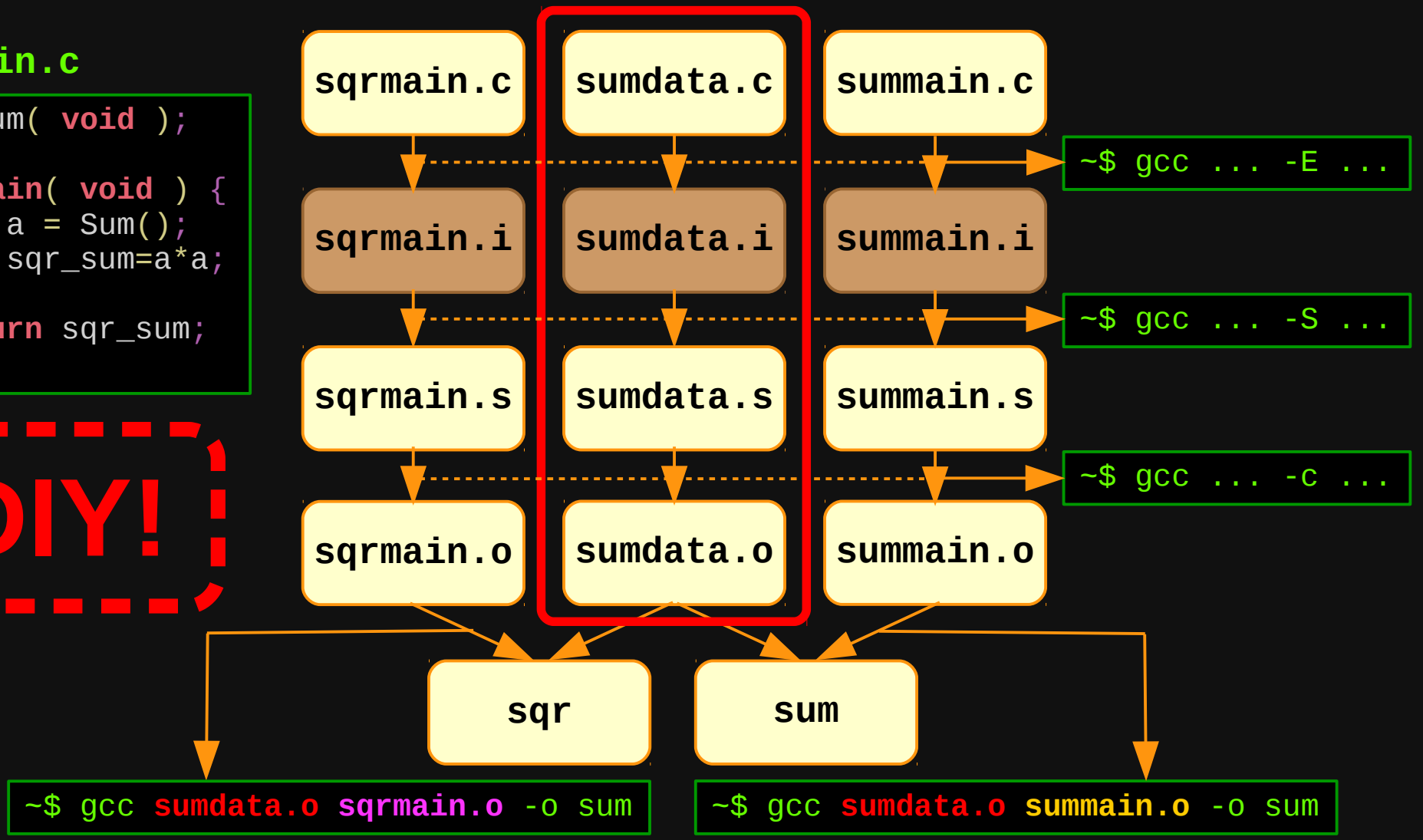
`.PHONY` indicates that the target `clean` is not related to a file called `clean` (the rule does not produce files!).
Thus, every time `clean` is called, it will be executed!

Reusing the **Sum( )** Function

sqrmain.c

```
int Sum( void );

int main( void ) {
  int a = Sum();
  int sqr_sum=a*a;

  return sqr_sum;
}
```

DIY!



Modifying the **Sum( )** Definition

Now we want **Sum( )** to iterate over all the elements of **x**, or only over its odd elements, according to an new input parameter

sumdata.c

```
int x[5] = { 1, 2, 3, 4, 5 };

int Sum( int sum_even ) {
    int i;
    int acc = 0;

    for ( i = 0; i < 5; i++ )
        if ((!sum_even) || (!(x[i] % 2)))
            acc += x[i];

    return acc;
}
```

We will need to modify the summain.c source accordingly.

summain.c

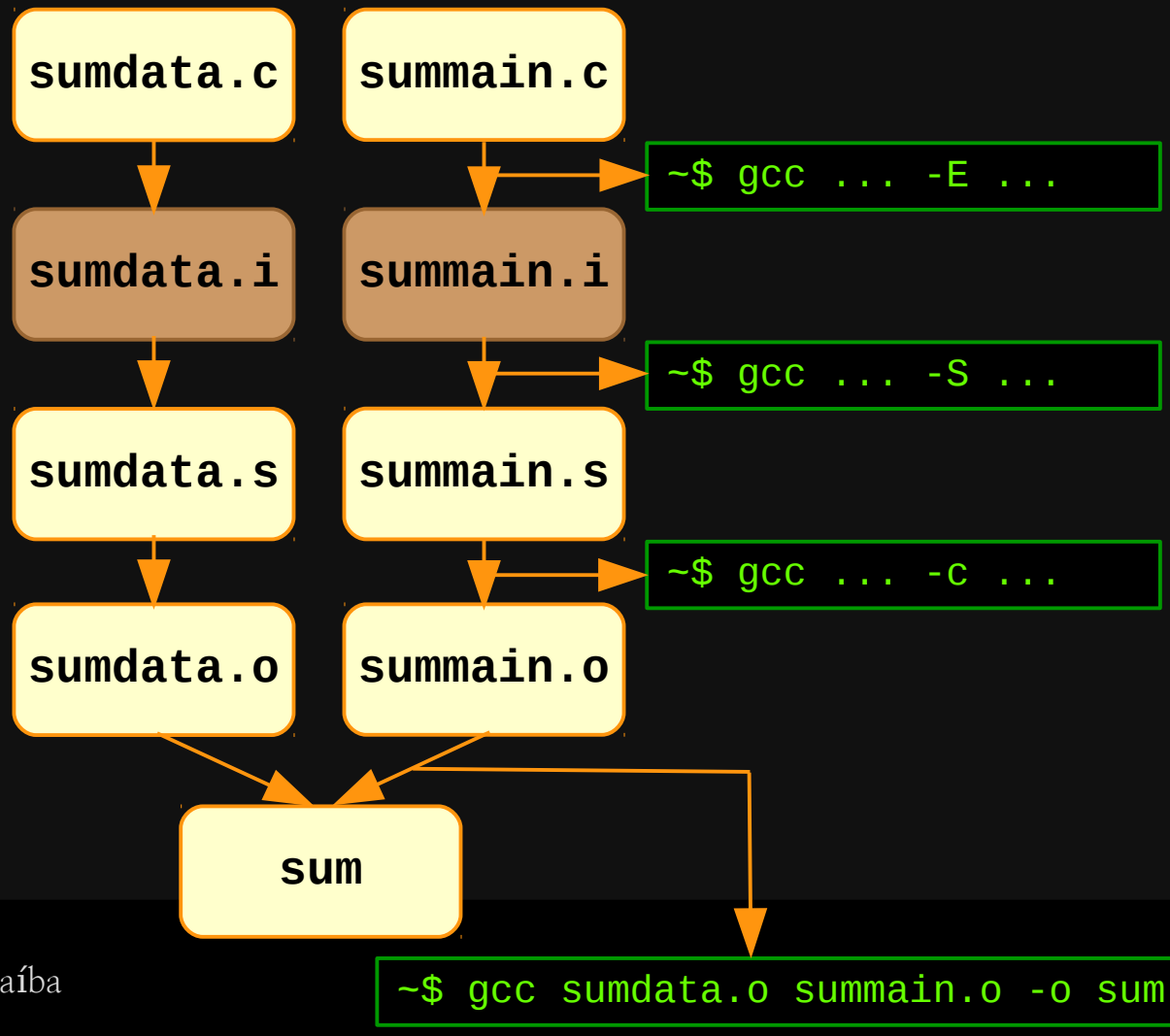
```
int Sum( int sum_even );

int main( void ) {
    int sum_value = Sum( 1 );

    return sum_value;
}
```

Building the New Program

Since we have changed both **sumdata.c** and **summain.c**, we have to rebuild everything from scratch

Execute the program and check the results...

Everything is Ok!

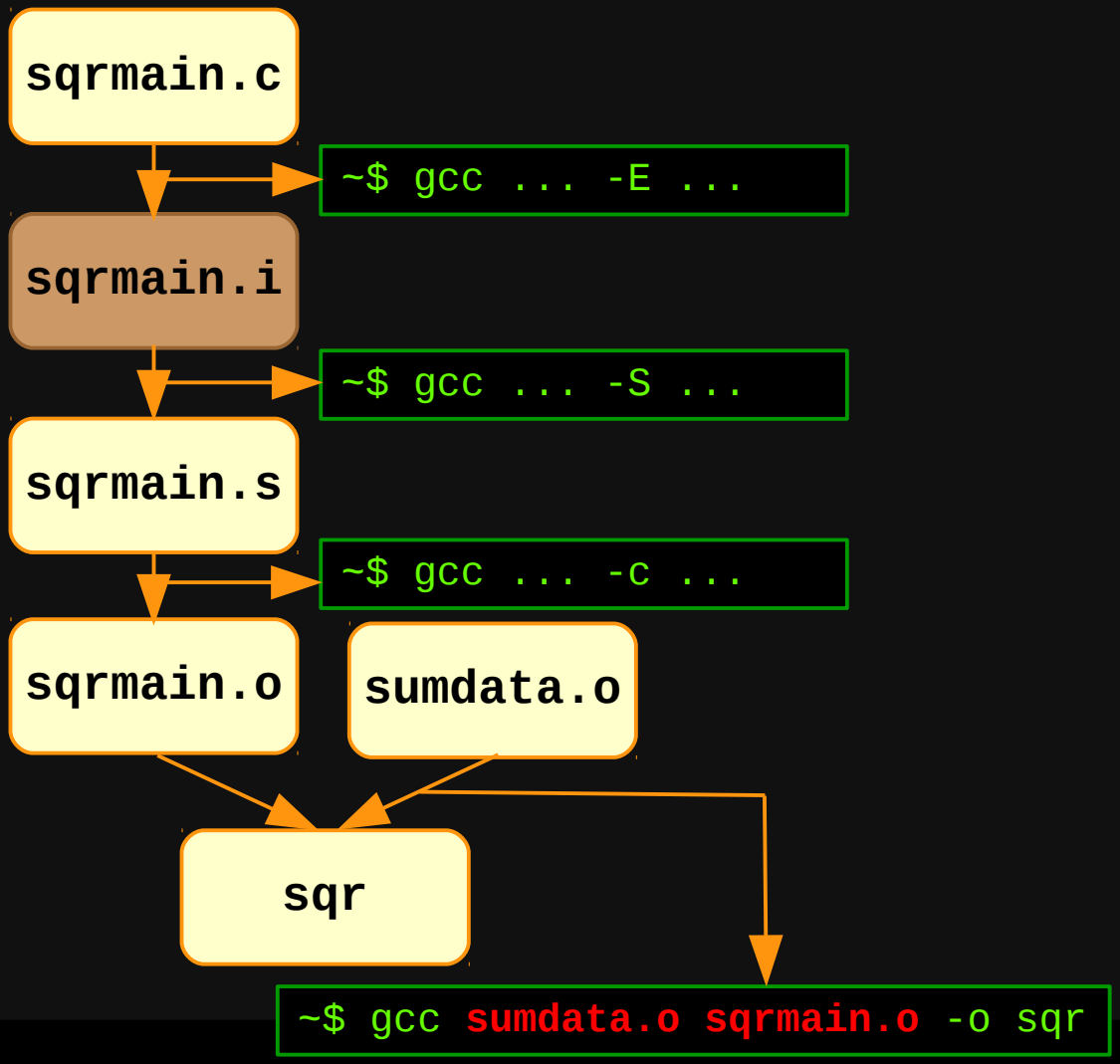


How about **sqrmain.c**?

Once we already have **sumdata.o**, let's rebuild only **sqrmain.c**

We **have not** updated the **sqrmain.c** source and we've **successfully built it !!!**

However, the result generated by **sqr** is **weird**. Could you **point out** what **is wrong**?



Fixing **sqr**

In order to fix **sqr**, we have to change its source code:

- Update the **Sum()** declaration (this allows the compiler to correctly detect the error in the function call!).
- Change the call to **Sum()** in **main()**, such that it receives the new input parameter.

sqrmain.c

```
int Sum( void );

int main( void ) {
  int a = Sum();
  int sqr_sum=a*a;

  return sqr_sum;
}
```

sqrmain.c

```
int Sum( int sum_even );

int main( void ) {
  int a = Sum( 1 ); // or Sum( 0 );
  int sqr_sum=a*a;

  return sqr_sum;
}
```

Header files make it easier to keep **prototypes up to date**!

Header Files

Adding a header file for **sumdata.c**:

- The original **.c** file will contain object definitions.
- The header will contain declarations.

sumdata.c

```c
int x[5] = { 1, 2, 3, 4, 5 };

int Sum( int sum_even ) {
    int i;
    int acc = 0;

    for ( i = 0; i < 5; i++ )
        if ((!sum_even) || (!(x[i] % 2)))
            acc += x[i];

    return acc;
}
```

sumdata.h

```c
int Sum( int sum_even );
```

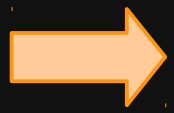
Now, any program wishing to use **Sum()** must **#include "sumdata.h"**

Header Files

Including **sumdata.h** in the previous programs

summain.c

```
int Sum( int sum_even );

int main( void ) {
    int sum_value = Sum( 1 );

    return sum_value;
}
```



summain.c

```
#include “sumdata.h”

int main( void ) {
    int sum_value = Sum( 1 );

    return sum_value;
}
```

sqrmain.c

```
int Sum( int sum_even );

int main( void ) {
  int a = Sum( 1 ); // or Sum( 0 );
  int sqr_sum=a*a;

  return sqr_sum;
}
```

sqrmain.c

```
#include “sumdata.h”

int main( void ) {
  int a = Sum( 1 ); // or Sum( 0 );
  int sqr_sum=a*a;

  return sqr_sum;
}
```

Header Files

- The **#include** preprocessor directive is just textual substitution.
- Example

sumdata.h

```
int Sum( int sum_even );
```

summain.c

```
#include “sumdata.h”

int main( void ) {
    int sum_value = Sum( 1 );

    return sum_value;
}
```

summain.i

```
# 1 "summain.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 1 "<command-line>" 2
# 1 "summain.c"
# 1 "sumdata.h" 1
int Sum( int sum_even );
# 2 "summain.c" 2

int main( void ) {
    int sum_value = Sum( 1 );

    return sum_value;
}
```

```
~$  gcc -Wall -Werror -E summain.c > summain.i
```

Who should `#include` the Headers?

- Source code that makes use of objects defined elsewhere (in other source files) should include the corresponding headers.

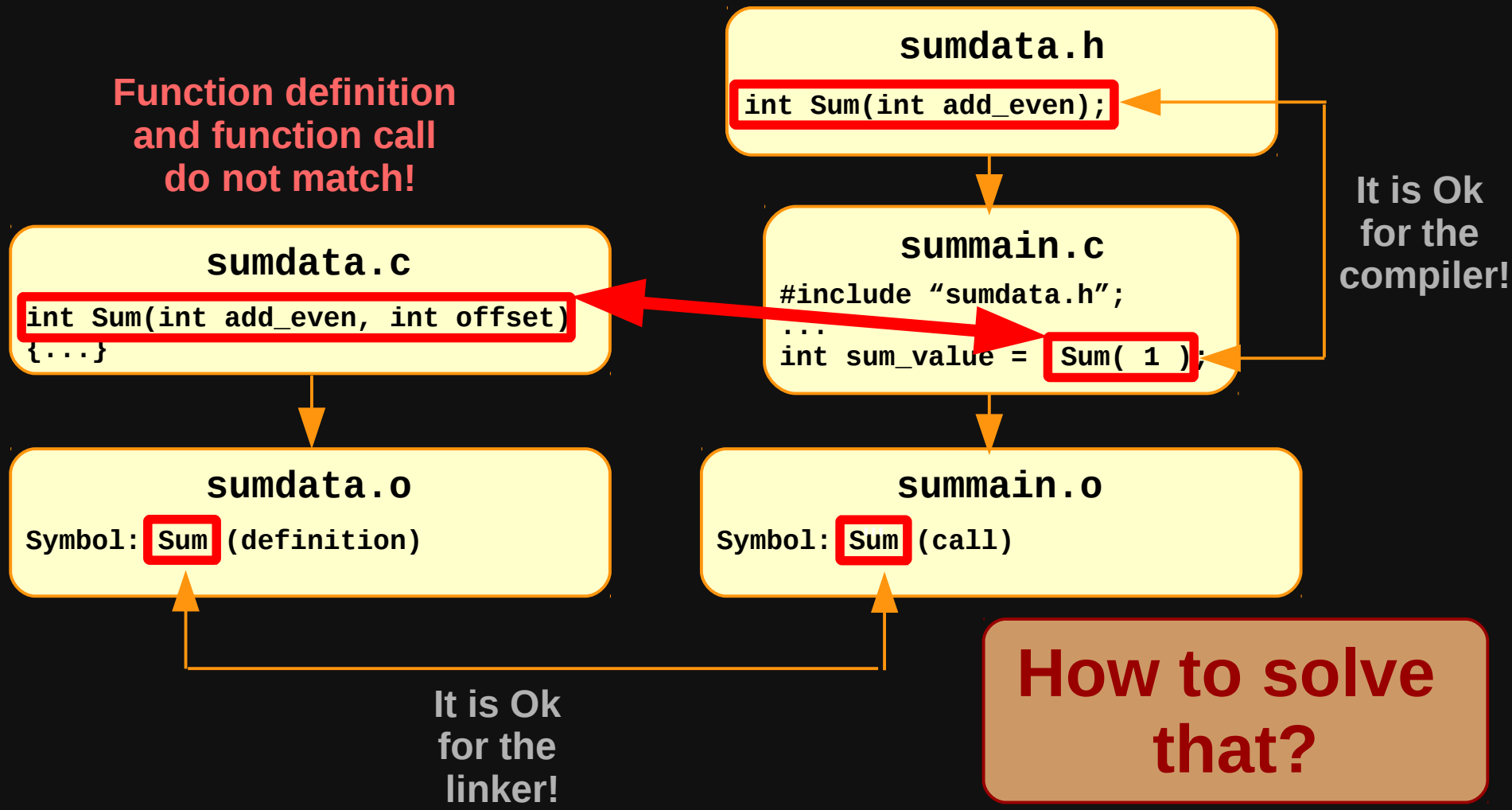
What if later changes to object definitions (e.g **parameter list, return type**, etc.) are not properly reflected by their headers?

An (**eventually malfunctioning**) executable could still be **successfully built**, because:

- The **compiler** won't detect any difference between the (unchanged) **`#included`** function **declaration** and the function **call**.
- The **linker** will look only at the function identifiers and, since they have not changed, no problems will be detected.

Who should `#include` the Headers?

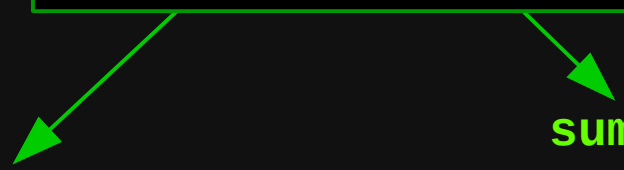
Changing definition (not the header!)



Inc. Declaration in the Definition File

sumdata.h

```
int Sum( int sum_even );
```



sumdata.c

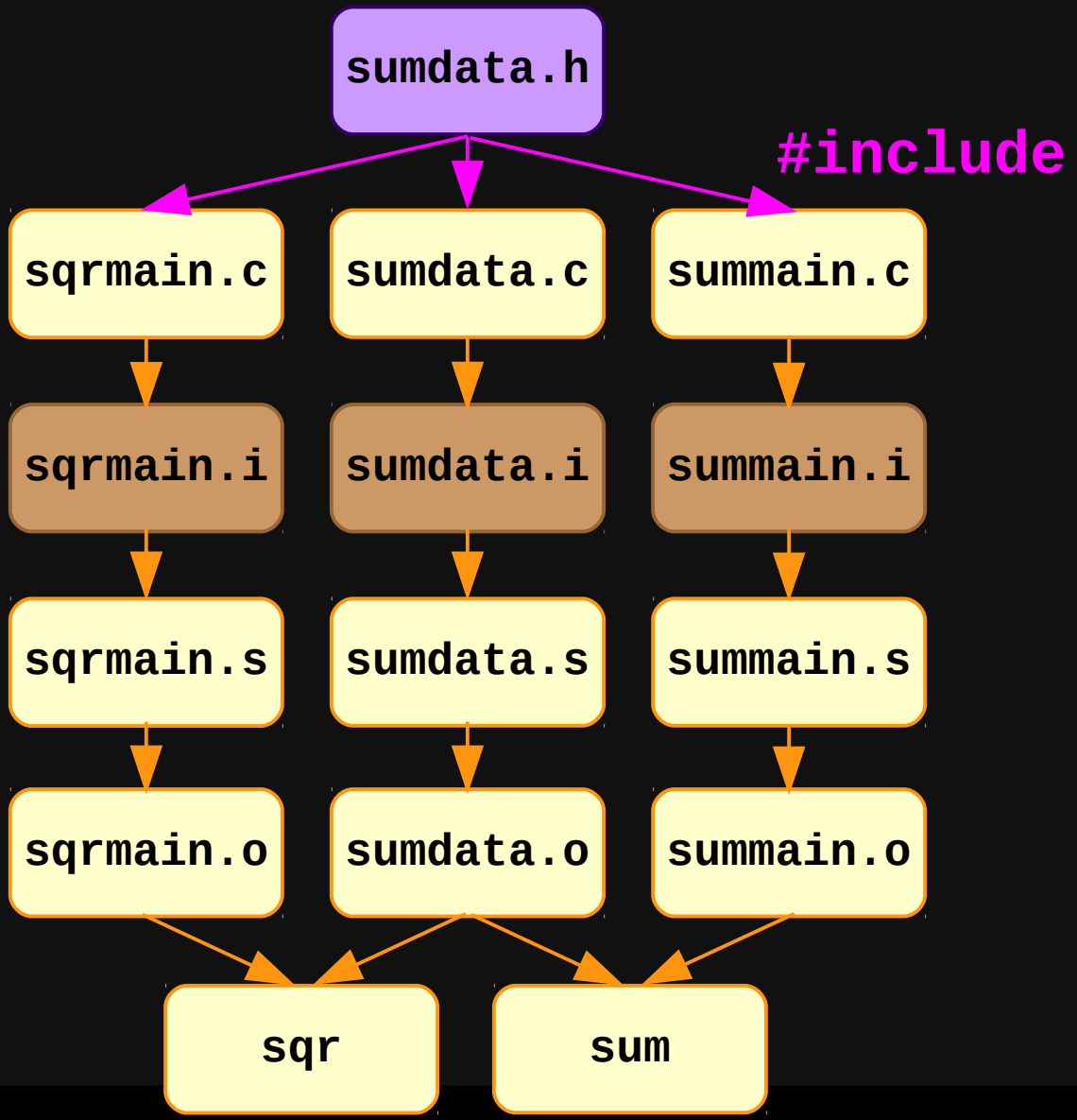
```
#include “sumdata.h”

int x[5] = { 1, 2, 3, 4, 5 };

int Sum( int sum_even ) {
    int i;
    int acc = 0;

    for ( i = 0; i < 5; i++ )
        if ((!sum_even) || (!(x[i] % 2)))
            acc += x[i];

    return acc;
}
```

summain.c

```
#include “sumdata.h”

int main( void ) {
    int sum_value = Sum( 1 );

    return sum_value;
}
```

Now, the compiler can check for discrepancies between the **function declaration**, **function definition** and **function call**.

Building **sum** and **sqr** with **#include**



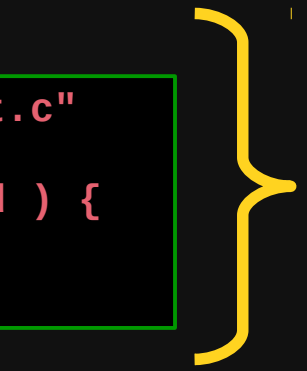
Observe the example below...

test.c

```
int Dummy( void )
{ }
```

main.c

```
#include "test.c"

int main( void ) {
    return 0;
}
```



Program source files.

Build commands:

```
~$ gcc -c main.c
~$ gcc -c test.c
~$ gcc main.o test.o -o dummyprog
```

What is happening here?

Linkage

- We can make several identifier **declarations** refer to the **same** object / function through **linkage**.
- There are three types of linkage:
  - External linkage.
  - Internal linkage.
  - None.

External Linkage

- The variable / function is **defined outside** the current **translation unit**.
- Given a **set** of translation units, "*each **declaration** of a particular **identifier** with **external linkage** denotes the **same object** or **function***".

sumdata.c

```c
#include "sumdata.h"

int x[5] = { 1, 2, 3, 4, 5 };

int Sum( int sum_even ) {
    int i;
    int acc = 0;
    {...}
    return acc;
}
```

```
~$ nm sumdata.o
0000000000000000 T Sum
0000000000000000 D x
```

x has external linkage

C99 Spec.

summain.c

```c
#include "sumdata.h"

extern int x[];

int main( void ) {
    {...}
}
```

x can be directly accessed through the `extern` qualifier.

Internal Linkage

- The variable / function is **defined inside** the current **translation unit**, and **is not visible** from outside.

sumdata.c

```
#include “sumdata.h”

static int x[5] = { 1, 2, 3, 4, 5 };

int Sum( int sum_even ) {
    int i;
    int acc = 0;
    {...}
    return acc;
}
```

x has internal linkage

```
~$ nm sumdata.o
0000000000000000 T Sum
0000000000000000 d x
```

summain.c

```
#include “sumdata.h”

extern int x[];

int main( void ) {
    {...}
}
```

x is not visible outside its TU.

```
~$ gcc -Wall -Werror summain.o sumdata.o -o sum
summain.o: In function `main':
summain.c:(.text+0x1c): undefined reference to `x'
collect2: error: ld returned 1 exit status
```