Introduction to Programming

Week – *11*, Lecture – *2*

Assorted Topics in C – Part 2

SAURABH SRIVASTAVA

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

IIT KANPUR

The C Preprocessor - 1/5

The C language implementations come with a component called the *Preprocessor*

The C Preprocessor - 1/5

The C language implementations come with a component called the *Preprocessor*

You have been using it since your first program

- You use the Preprocessor every time you write a `#include` statement

The C Preprocessor - 1/5

The C language implementations come with a component called the *Preprocessor*

You have been using it since your first program

- You use the Preprocessor every time you write a `#include` statement

The Preprocessor, as the name suggests, performs some “pre-processing”…

- … before your program is handled by the compiler

The C Preprocessor - 1/5

The C language implementations come with a component called the *Preprocessor*

You have been using it since your first program

- You use the Preprocessor every time you write a `#include` statement

The Preprocessor, as the name suggests, performs some “pre-processing”…

- … before your program is handled by the compiler

The `#include` directive of the Preprocessor allows you to add source code from other files

The C Preprocessor - 1/5

The C language implementations come with a component called the *Preprocessor*

You have been using it since your first program

- You use the Preprocessor every time you write a `#include` statement

The Preprocessor, as the name suggests, performs some “pre-processing”…

- … before your program is handled by the compiler

The `#include` directive of the Preprocessor allows you to add source code from other files

You may have also seen the `#define` directive for defining constants

The C Preprocessor - 1/5

The C language implementations come with a component called the *Preprocessor*

You have been using it since your first program

- You use the Preprocessor every time you write a `#include` statement

The Preprocessor, as the name suggests, performs some "pre-processing"…

- … before your program is handled by the compiler

The `#include` directive of the Preprocessor allows you to add source code from other files

You may have also seen the `#define` directive for defining constants

Just to be clear – the compiler never sees these directives

- They are replaced by "appropriate code" before the compilation starts

The C Preprocessor - 1/5

The C language implementations come with a component called the *Preprocessor*

You have been using it since your first program

- You use the Preprocessor every time you write a `#include` statement

The Preprocessor, as the name suggests, performs some "pre-processing"…

- … before your program is handled by the compiler

The `#include` directive of the Preprocessor allows you to add source code from other files

You may have also seen the `#define` directive for defining constants

Just to be clear – the compiler never sees these directives

- They are replaced by "appropriate code" before the compilation starts

But prior to discussing these directives, let us talk about splitting our code into multiple files

Detour - Splitting C code into multiple files

It is seldom a good idea to put too much code in one code file

- Irrespective of the language you are using
- At some point in time, you will have to split your code across multiple files

Detour - Splitting C code into multiple files

It is seldom a good idea to put too much code in one code file

- Irrespective of the language you are using
- At some point in time, you will have to split your code across multiple files

Detour - Splitting C code into multiple files

It is seldom a good idea to put too much code in one code file

- Irrespective of the language you are using
- At some point in time, you will have to split your code across multiple files

We saw a simple example of doing that in Week 8

- Remember the `Car.h` file which contained the template definition of a structure?

Detour - Splitting C code into multiple files

It is seldom a good idea to put too much code in one code file

- Irrespective of the language you are using
- At some point in time, you will have to split your code across multiple files

We saw a simple example of doing that in Week 8

- Remember the `Car.h` file which contained the template definition of a structure?

In fact, if you are creating a C library, it is a common practice to

Detour - Splitting C code into multiple files

It is seldom a good idea to put too much code in one code file

- Irrespective of the language you are using
- At some point in time, you will have to split your code across multiple files

We saw a simple example of doing that in Week 8

- Remember the `Car.h` file which contained the template definition of a structure?

In fact, if you are creating a C library, it is a common practice to

- Collect *declarations* of supplied functions in one or more header files (usually with extension `.h`)

Detour - Splitting C code into multiple files

It is seldom a good idea to put too much code in one code file

- Irrespective of the language you are using
- At some point in time, you will have to split your code across multiple files

We saw a simple example of doing that in Week 8

- Remember the `Car.h` file which contained the template definition of a structure?

In fact, if you are creating a C library, it is a common practice to

- Collect *declarations* of supplied functions in one or more header files (usually with extension `.h`)
- Write their *definitions* in one or more source files (usually with extension `.c`)

Detour - Splitting C code into multiple files

It is seldom a good idea to put too much code in one code file

- Irrespective of the language you are using
- At some point in time, you will have to split your code across multiple files

We saw a simple example of doing that in Week 8

- Remember the `Car.h` file which contained the template definition of a structure?

In fact, if you are creating a C library, it is a common practice to

- Collect *declarations* of supplied functions in one or more header files (usually with extension `.h`)
- Write their *definitions* in one or more source files (usually with extension `.c`)
- Provide only the object code (compiled binary code) of the definitions for "linking" with other applications

Detour - Splitting C code into multiple files

It is seldom a good idea to put too much code in one code file

- Irrespective of the language you are using
- At some point in time, you will have to split your code across multiple files

We saw a simple example of doing that in Week 8

- Remember the `Car.h` file which contained the template definition of a structure?

In fact, if you are creating a C library, it is a common practice to

- Collect *declarations* of supplied functions in one or more header files (usually with extension `.h`)
- Write their *definitions* in one or more source files (usually with extension `.c`)
- Provide only the object code (compiled binary code) of the definitions for "linking" with other applications

While using the library, the users are expected to

- Include the header file(s) in their source files, and
- Link the object code of the definitions (e.g. we did so with the `-lm` switch for using the math library)

The C Preprocessor - 2/5

The most commonly used Preprocessor directive is `#include`

The C Preprocessor - 2/5

The most commonly used Preprocessor directive is `#include`

The `#include` directive is used for “copying and pasting” contents of one file into another

The C Preprocessor - 2/5

The most commonly used Preprocessor directive is `#include`

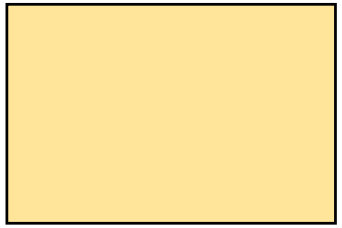
The `#include` directive is used for “copying and pasting” contents of one file into another

Thus, when the compiler gets the code for compilation, it will appear that the included code…

- … was a part of the source file it is being given to compile

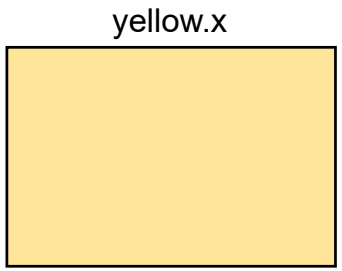
The `#include` directive

yellow.x



Assume that we have a file called `yellow.x`

The `#include` directive

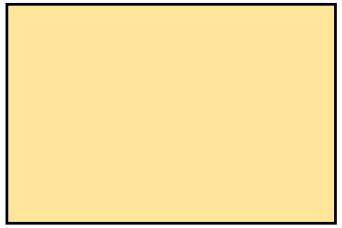


Assume that we have a file called `yellow.x`

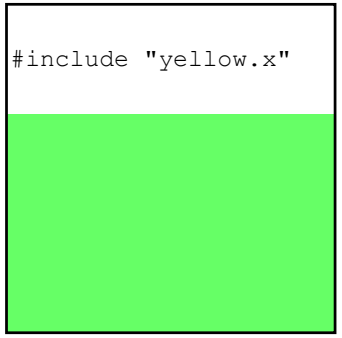
While there are no strict constraints on extensions, `.x` usually means `.c` or `.h`

The `#include` directive

yellow.x



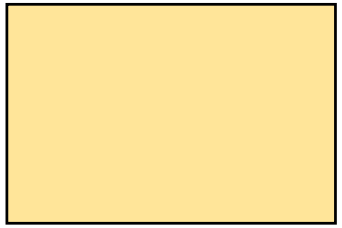
green.x



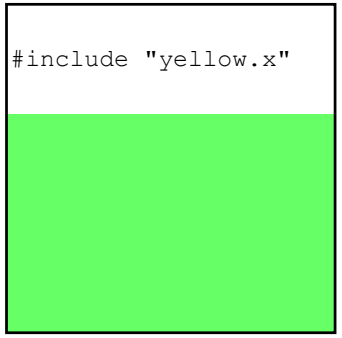
Assume that we include the file `yellow.x` in another file, `green.x`

The `#include` directive

yellow.x



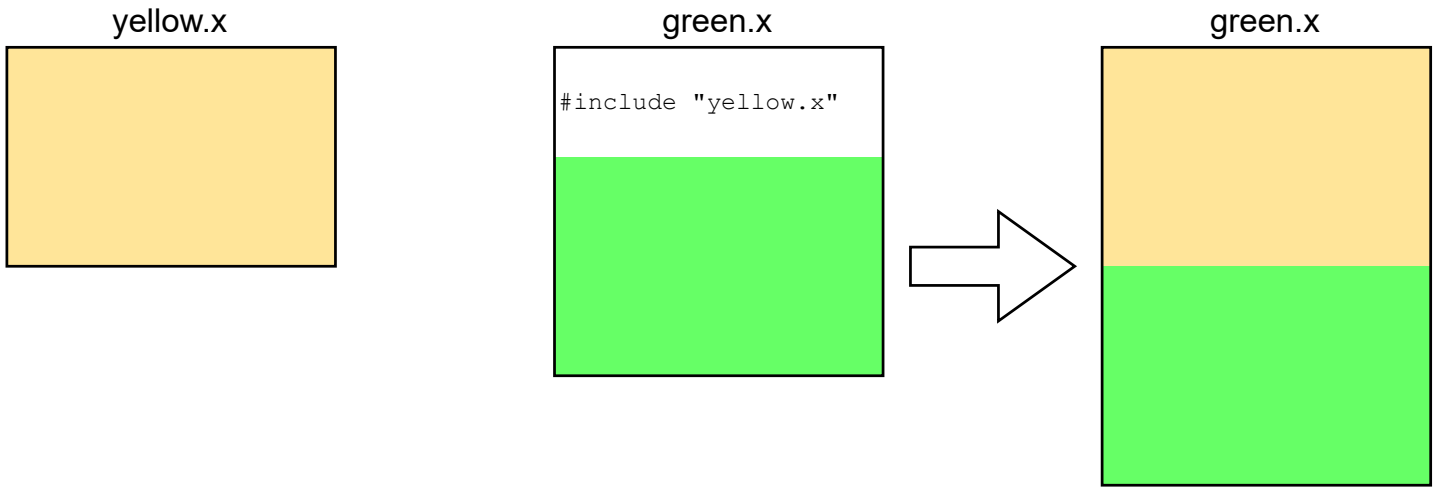
green.x



Assume that we include the file `yellow.x` in another file, `green.x`

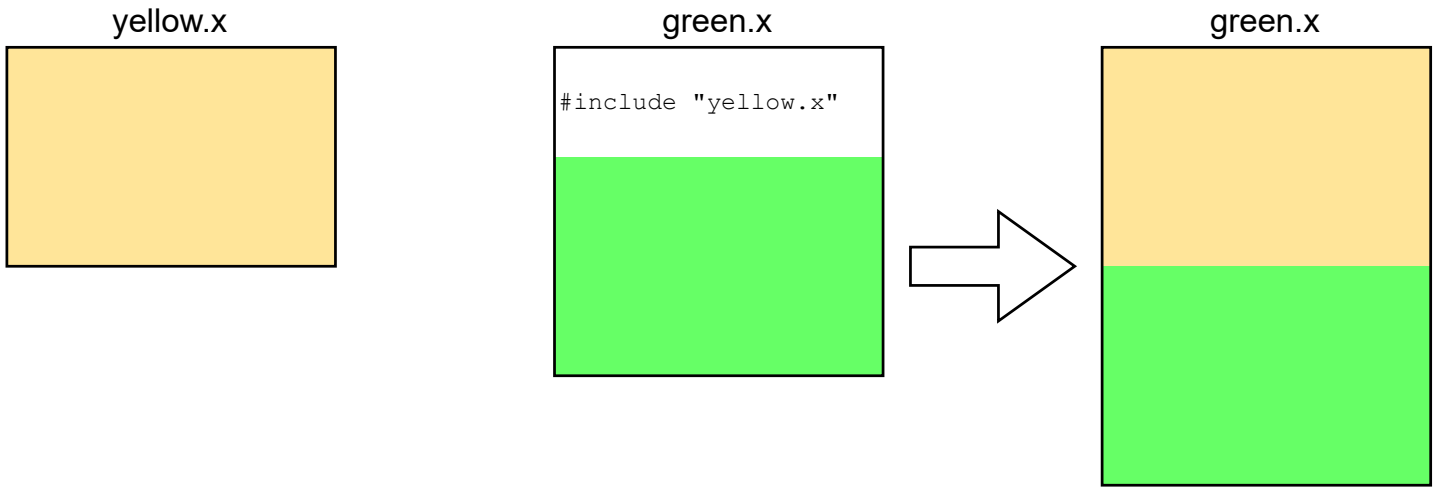
`green.x` may also have other code (usually making use of `yellow.x`)

The `#include` directive



When `green.x` is compiled, its code looks similar to what is shown here…

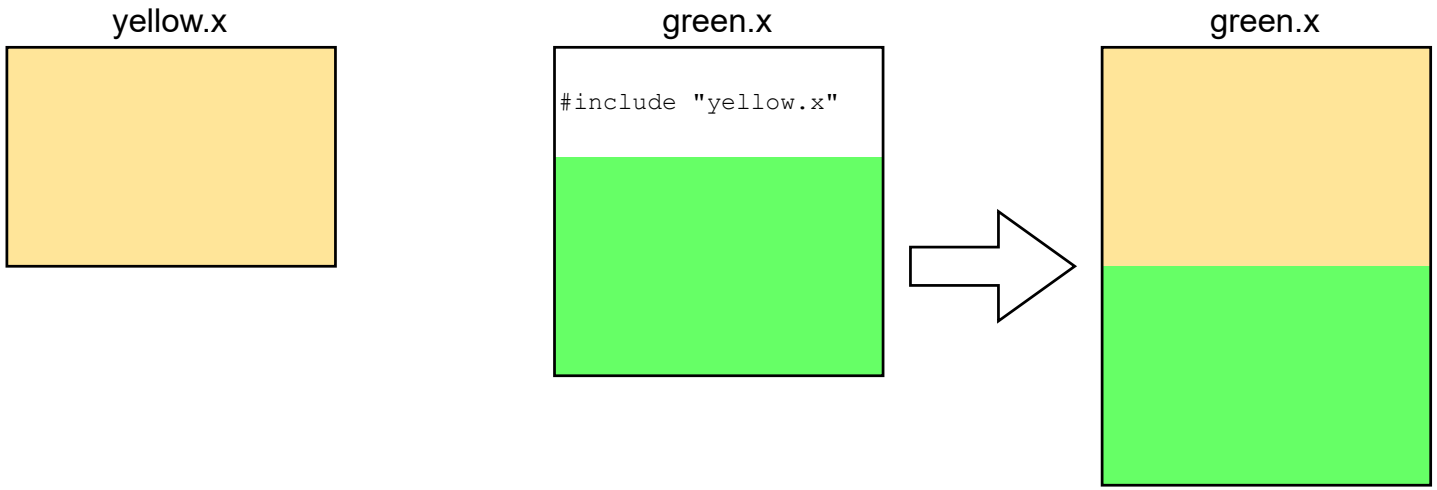
The `#include` directive



When `green.x` is compiled, its code looks similar to what is shown here…

… i.e., it appears as if the code of `yellow.x` is a part of `green.x`

The `#include` directive



When `green.x` is compiled, its code looks similar to what is shown here…

… i.e., it appears as if the code of `yellow.x` is a part of `green.x`

Although not accurate, this depicts the overall idea of `#include` directive

The C Preprocessor - 3/5

Another commonly used directive is the `#define` directive

The C Preprocessor - 3/5

Another commonly used directive is the `#define` directive

The `#define` directive is usually employed for three use cases

The C Preprocessor - 3/5

Another commonly used directive is the `#define` directive

The `#define` directive is usually employed for three use cases

- You can use it to define simple constants, like the size of an array

The C Preprocessor - 3/5

Another commonly used directive is the `#define` directive

The `#define` directive is usually employed for three use cases

- You can use it to define simple constants, like the size of an array
- You can use it to define what are commonly known as *macros* – replacements for simple functions

The C Preprocessor - 3/5

Another commonly used directive is the `#define` directive

The `#define` directive is usually employed for three use cases

- You can use it to define simple constants, like the size of an array
- You can use it to define what are commonly known as *macros* – replacements for simple functions
- They can be used for enabling *conditional preprocessing* – we'll come back to this later

The C Preprocessor - 3/5

Another commonly used directive is the `#define` directive

The `#define` directive is usually employed for three use cases

- You can use it to define simple constants, like the size of an array
- You can use it to define what are commonly known as *macros* – replacements for simple functions
- They can be used for enabling *conditional preprocessing* – we'll come back to this later

We have seen their use for creating simple constants

- For instance, statements like `#define SIZE_OF_ARRAY 10`

The C Preprocessor - 3/5

Another commonly used directive is the `#define` directive

The `#define` directive is usually employed for three use cases

- You can use it to define simple constants, like the size of an array
- You can use it to define what are commonly known as *macros* – replacements for simple functions
- They can be used for enabling *conditional preprocessing* – we'll come back to this later

We have seen their use for creating simple constants

- For instance, statements like `#define SIZE_OF_ARRAY 10`

We can also use `#define` directives to create macros

The C Preprocessor - 3/5

Another commonly used directive is the `#define` directive

The `#define` directive is usually employed for three use cases

- You can use it to define simple constants, like the size of an array
- You can use it to define what are commonly known as *macros* – replacements for simple functions
- They can be used for enabling *conditional preprocessing* – we'll come back to this later

We have seen their use for creating simple constants

- For instance, statements like `#define SIZE_OF_ARRAY 10`

We can also use `#define` directives to create macros

- Actually, `#define` works fairly similar to `#include` – it just does some "copying and pasting"

The C Preprocessor - 3/5

Another commonly used directive is the `#define` directive

The `#define` directive is usually employed for three use cases

- You can use it to define simple constants, like the size of an array
- You can use it to define what are commonly known as *macros* – replacements for simple functions
- They can be used for enabling *conditional preprocessing* – we'll come back to this later

We have seen their use for creating simple constants

- For instance, statements like `#define SIZE_OF_ARRAY 10`

We can also use `#define` directives to create macros

- Actually, `#define` works fairly similar to `#include` – it just does some "copying and pasting"
- We can use this "dumb" copying and pasting to our advantage…
- … and create what could be seen as some kind of "data type agnostic functions"

The C Preprocessor - 3/5

Another commonly used directive is the `#define` directive

The `#define` directive is usually employed for three use cases

- You can use it to define simple constants, like the size of an array
- You can use it to define what are commonly known as *macros* – replacements for simple functions
- They can be used for enabling *conditional preprocessing* – we'll come back to this later

We have seen their use for creating simple constants

- For instance, statements like `#define SIZE_OF_ARRAY 10`

We can also use `#define` directives to create macros

- Actually, `#define` works fairly similar to `#include` – it just does some "copying and pasting"
- We can use this "dumb" copying and pasting to our advantage…
- … and create what could be seen as some kind of "data type agnostic functions"

However, remember that this copying and pasting may have unexpected results too !!

The #define directive

```
...

#define square(x) x*x

...
```

Here, we are creating a macro called `square`

The `#define` directive

```
...

#define square(x) x*x

...
```

Here, we are creating a macro called `square`

This is analogous to a function that can return the square of a passed number

The #define directive

```
...

#define square(x) x*x

...
```

This is how we can use this macro in the code

```
...

sq1 = square(5);

...

sq2 = square(sq1);

...
```

The #define directive

```
...

#define square(x) x*x

...
```

This is how we can use this macro in the code

```
...

sq1 = square(5);

...

sq2 = square(sq1);

...
```

It looks identical to a function call

The #define directive

```
...

#define square(x) x*x

...
```

```
...

sq1 = square(5);

...

sq2 = square(sq1);

...
```

This is how we can use this macro in the code

It looks identical to a function call

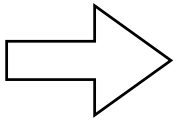
However, all that is going to happen here, is some “copying and pasting”

The #define directive

```
...

#define square(x) x*x

...
```

The compiler essentially sees the code as shown here

```
...

sq1 = square(5);

...

sq2 = square(sq1);

...
```



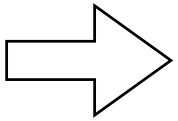
```
...

sq1 = 5*5;

...

sq2 = sq1*sq1;

...
```

The #define directive

```
...

#define square(x) x*x

...
```

The compiler essentially sees the code as shown here

```
...

sq1 = square(5);

...

sq2 = square(sq1);

...
```



```
...

sq1 = 5*5;

...

sq2 = sq1*sq1;

...
```

The advantage here is that you can pass constants, variables or expressions

The #define directive

```
...

#define square(x) x*x

...
```

The compiler essentially sees the code as shown here

```
...

sq1 = square(5);

...

sq2 = square(sq1);

...
```

```
...

sq1 = 5*5;

...

sq2 = sq1*sq1;

...
```

The advantage here is that you can pass constants, variables or expressions

There is an advantage here too – the arguments need not be of a fixed type !!

The #define directive

```
...

#define square(x) x*x

...
```

The compiler essentially sees the code as shown here

```
...

sq1 = square(5);

...

sq2 = square(sq1);

...
```

```
...

sq1 = 5*5;

...

sq2 = sq1*sq1;

...
```

The advantage here is that you can pass constants, variables or expressions

There is an advantage here too – the arguments need not be of a fixed type !!

As long as the copying and pasting *makes sense*, it is fine

The C Preprocessor - 4/5

Till now, the examples of preprocessing that we saw, though helpful, were really not impressive

- Basically, we may feel that the preprocessing is nothing more than “dumb” copying and pasting

The C Preprocessor - 4/5

Till now, the examples of preprocessing that we saw, though helpful, were really not impressive

- Basically, we may feel that the preprocessing is nothing more than “dumb” copying and pasting
- But there is more to preprocessing than that – you can write some *conditions* too

The C Preprocessor - 4/5

Till now, the examples of preprocessing that we saw, though helpful, were really not impressive

- Basically, we may feel that the preprocessing is nothing more than “dumb” copying and pasting
- But there is more to preprocessing than that – you can write some *conditions* too

For instance, the `#if` directive can evaluate an expression (which must produce an integer)

The C Preprocessor - 4/5

Till now, the examples of preprocessing that we saw, though helpful, were really not impressive

- Basically, we may feel that the preprocessing is nothing more than “dumb” copying and pasting
- But there is more to preprocessing than that – you can write some *conditions* too

For instance, the `#if` directive can evaluate an expression (which must produce an integer)

- Any preprocessing code added after that, is executed only if the condition evaluates to true

The C Preprocessor - 4/5

Till now, the examples of preprocessing that we saw, though helpful, were really not impressive

- Basically, we may feel that the preprocessing is nothing more than “dumb” copying and pasting
- But there is more to preprocessing than that – you can write some *conditions* too

For instance, the `#if` directive can evaluate an expression (which must produce an integer)

- Any preprocessing code added after that, is executed only if the condition evaluates to true
- As an example, consider the following code

```
#if !defined(SIZE_OF_ARRAY)
    #define SIZE_OF_ARRAY 10
#endif
```

The C Preprocessor - 4/5

Till now, the examples of preprocessing that we saw, though helpful, were really not impressive

- Basically, we may feel that the preprocessing is nothing more than “dumb” copying and pasting
- But there is more to preprocessing than that – you can write some *conditions* too

For instance, the `#if` directive can evaluate an expression (which must produce an integer)

- Any preprocessing code added after that, is executed only if the condition evaluates to true
- As an example, consider the following code

```
#if !defined(SIZE_OF_ARRAY)
    #define SIZE_OF_ARRAY 10
#endif
```

- Here, if (and only if) the `SIZE_OF_ARRAY` constant has not been defined, a definition is provided

The C Preprocessor - 4/5

Till now, the examples of preprocessing that we saw, though helpful, were really not impressive

- Basically, we may feel that the preprocessing is nothing more than "dumb" copying and pasting
- But there is more to preprocessing than that – you can write some *conditions* too

For instance, the `#if` directive can evaluate an expression (which must produce an integer)

- Any preprocessing code added after that, is executed only if the condition evaluates to true
- As an example, consider the following code

```
#if !defined(SIZE_OF_ARRAY)
    #define SIZE_OF_ARRAY 10
#endif
```

- Here, if (and only if) the `SIZE_OF_ARRAY` constant has not been defined, a definition is provided
- The `defined(X)` expression returns 1, if `X` has been defined somewhere prior to this statement

The C Preprocessor - 4/5

Till now, the examples of preprocessing that we saw, though helpful, were really not impressive

- Basically, we may feel that the preprocessing is nothing more than "dumb" copying and pasting
- But there is more to preprocessing than that – you can write some *conditions* too

For instance, the `#if` directive can evaluate an expression (which must produce an integer)

- Any preprocessing code added after that, is executed only if the condition evaluates to true
- As an example, consider the following code

```
#if !defined(SIZE_OF_ARRAY)
    #define SIZE_OF_ARRAY 10
#endif
```

- Here, if (and only if) the `SIZE_OF_ARRAY` constant has not been defined, a definition is provided
- The `defined(X)` expression returns 1, if `X` has been defined somewhere prior to this statement
- The `#endif` directive acts similar to braces with blocks associated with the `if` conditions

The C Preprocessor - 4/5

Till now, the examples of preprocessing that we saw, though helpful, were really not impressive

- Basically, we may feel that the preprocessing is nothing more than "dumb" copying and pasting
- But there is more to preprocessing than that – you can write some *conditions* too

For instance, the `#if` directive can evaluate an expression (which must produce an integer)

- Any preprocessing code added after that, is executed only if the condition evaluates to true
- As an example, consider the following code

```
#if !defined(SIZE_OF_ARRAY)
    #define SIZE_OF_ARRAY 10
#endif
```

- Here, if (and only if) the `SIZE_OF_ARRAY` constant has not been defined, a definition is provided
- The `defined(X)` expression returns 1, if `X` has been defined somewhere prior to this statement
- The `#endif` directive acts similar to braces with blocks associated with the `if` conditions
- In fact, there are `#elif` and `#else` directives as well, to complete the analogy…
- … with the `else if` and `else` constructs

The C Preprocessor - 5/5

The type of checks we saw are often required to mitigate the problem of multiple inclusion

The C Preprocessor - 5/5

The type of checks we saw are often required to mitigate the problem of multiple inclusion

In fact, It is so common, that there are shortcut directives for this use – `#ifndef` and `#ifdef`

The C Preprocessor - 5/5

The type of checks we saw are often required to mitigate the problem of multiple inclusion

In fact, It is so common, that there are shortcut directives for this use – `#ifndef` and `#ifdef`

- Actually, `#if !defined(X)` is equivalent to writing `#ifndef X`
- Similarly, `#if defined (X)` is equivalent to writing `#ifdef X`

The C Preprocessor - 5/5

The type of checks we saw are often required to mitigate the problem of multiple inclusion

In fact, It is so common, that there are shortcut directives for this use – `#ifndef` and `#ifdef`

- Actually, `#if !defined(X)` is equivalent to writing `#ifndef X`
- Similarly, `#if defined (X)` is equivalent to writing `#ifdef X`

The multiple inclusion problem occurs, when a piece of code gets included multiple times…

- … within the same compilation unit

The C Preprocessor - 5/5

The type of checks we saw are often required to mitigate the problem of multiple inclusion

In fact, It is so common, that there are shortcut directives for this use – `#ifndef` and `#ifdef`

- Actually, `#if !defined(X)` is equivalent to writing `#ifndef X`
- Similarly, `#if defined (X)` is equivalent to writing `#ifdef X`

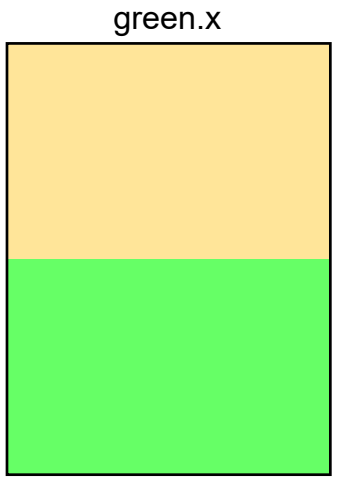
The multiple inclusion problem occurs, when a piece of code gets included multiple times…

- … within the same compilation unit

This could happen if the same file (usually a header file) gets included into multiple code files…

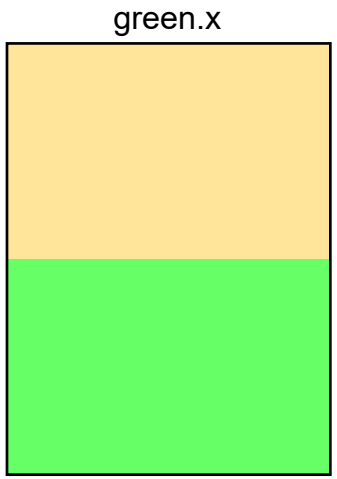
- … and then all these code files are compiled together to build an application

The multiple inclusion problem



Remember that in our example, we included `yellow.x` in `green.x`

The multiple inclusion problem

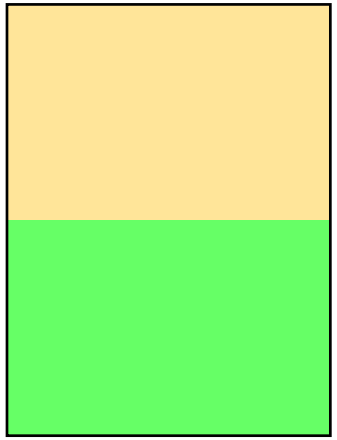


Remember that in our example, we included `yellow.x` in `green.x`

So, in effect, this is how `green.x` looks like during compilation

The multiple inclusion problem

green.x



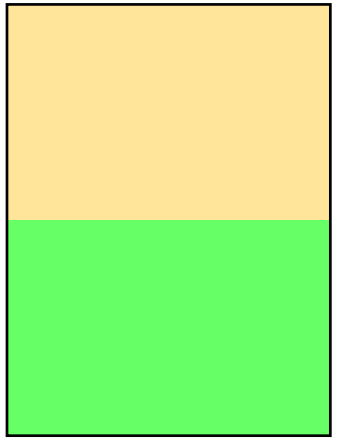
red.x

```
#include "green.x"

#include "yellow.x"
```

Now assume that we have another code file called `red.x`

The multiple inclusion problem

green.x



red.x

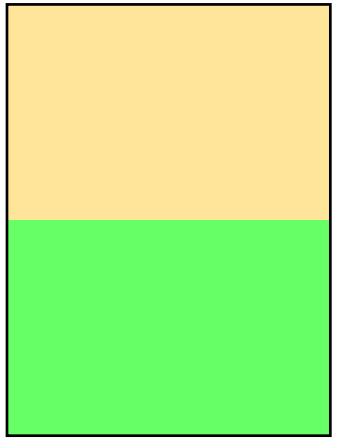
```
#include "green.x"

#include "yellow.x"
```

Now assume that we have another code file called `red.x`

Assume that If we are using something from both `green.x` and `yellow.x`

The multiple inclusion problem

green.x



red.x

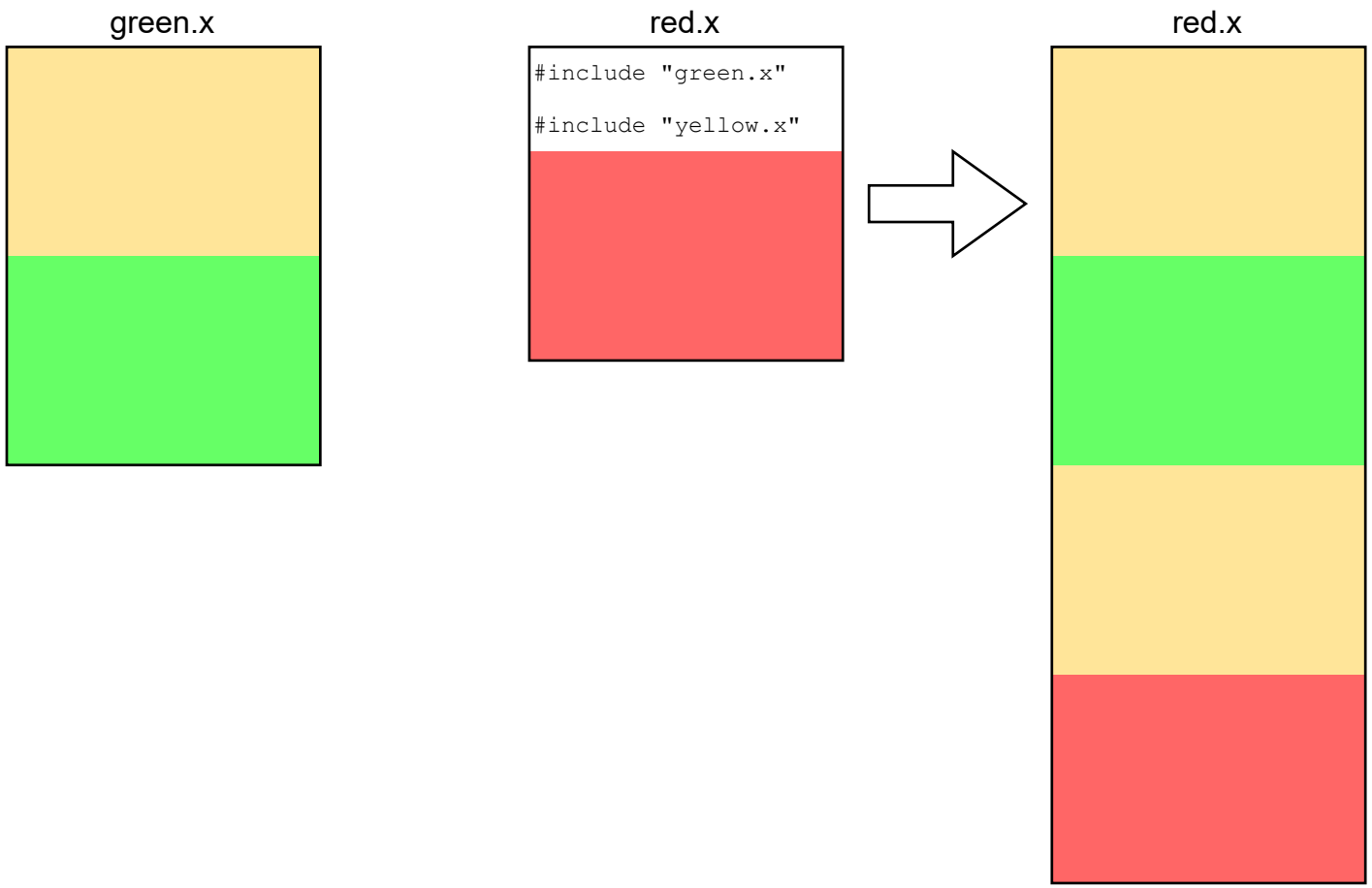
```
#include "green.x"

#include "yellow.x"
```

Now assume that we have another code file called `red.x`

Assume that If we are using something from both `green.x` and `yellow.x`

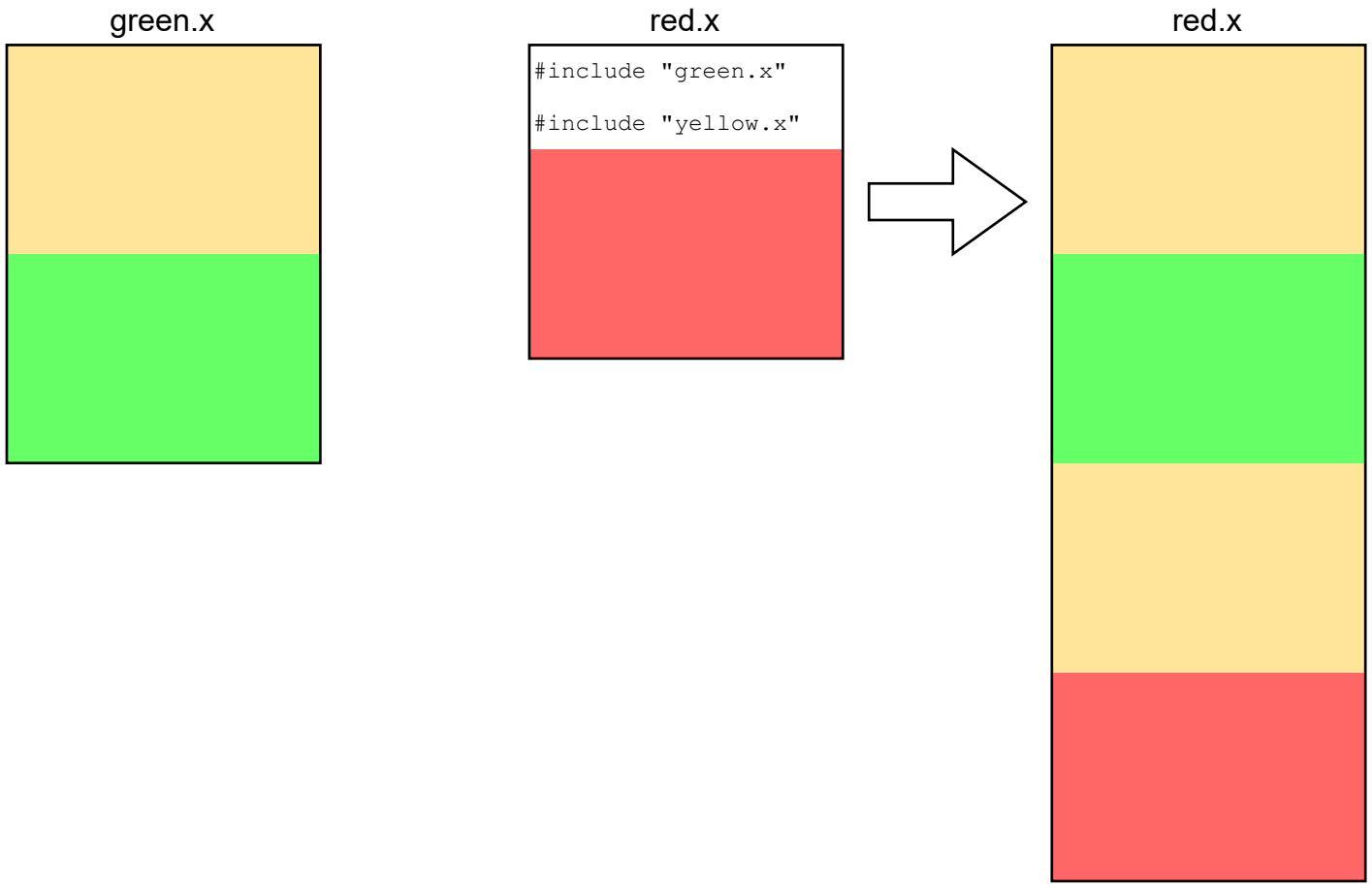
We may casually include both files in `red.x` as shown here

The multiple inclusion problem



But since `#include` simply does a copy and paste, we have a problem !!

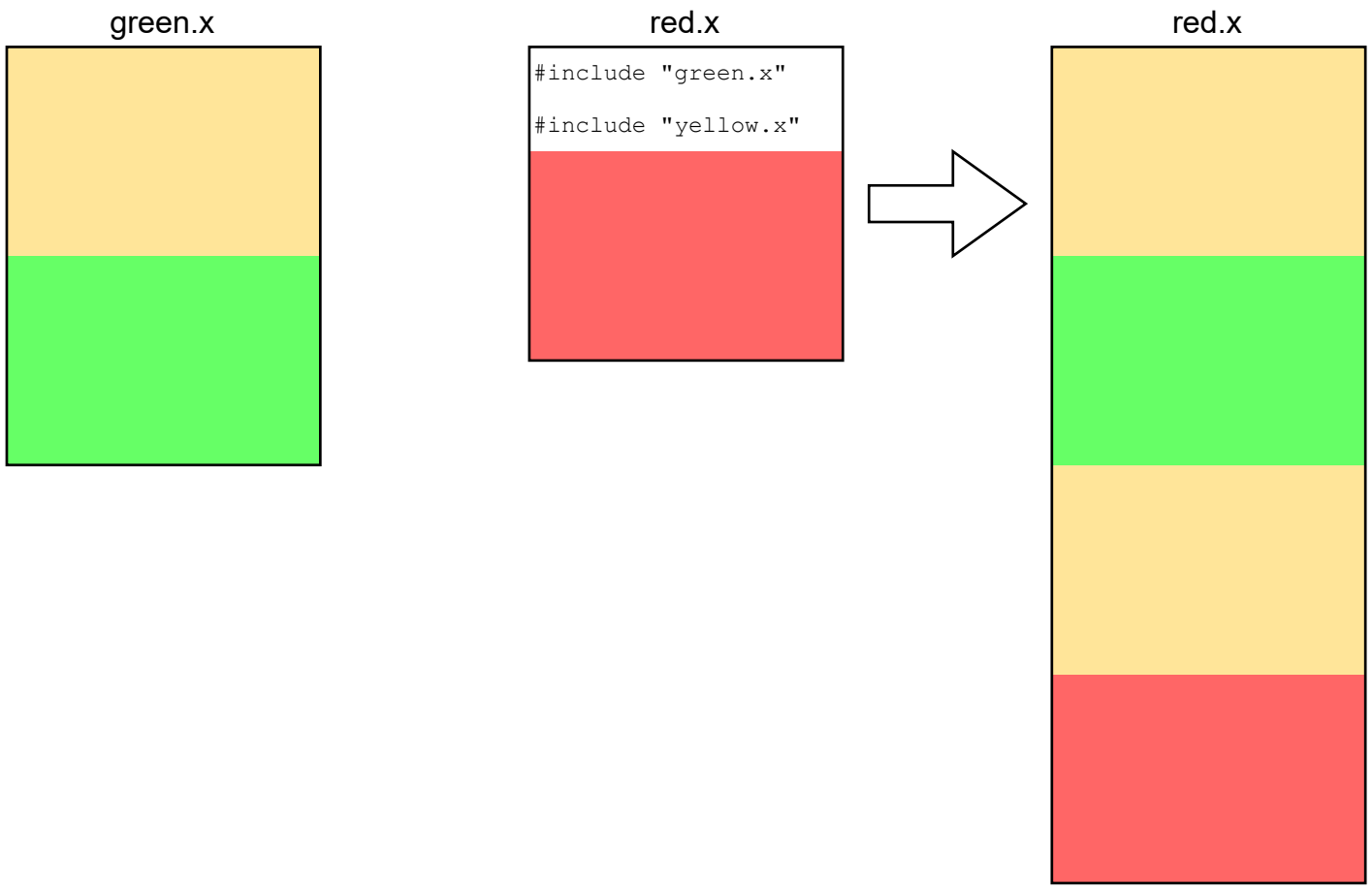
The multiple inclusion problem



But since `#include` simply does a copy and paste, we have a problem !!

The contents of `yellow.x` will be included twice in `red.x`

The multiple inclusion problem

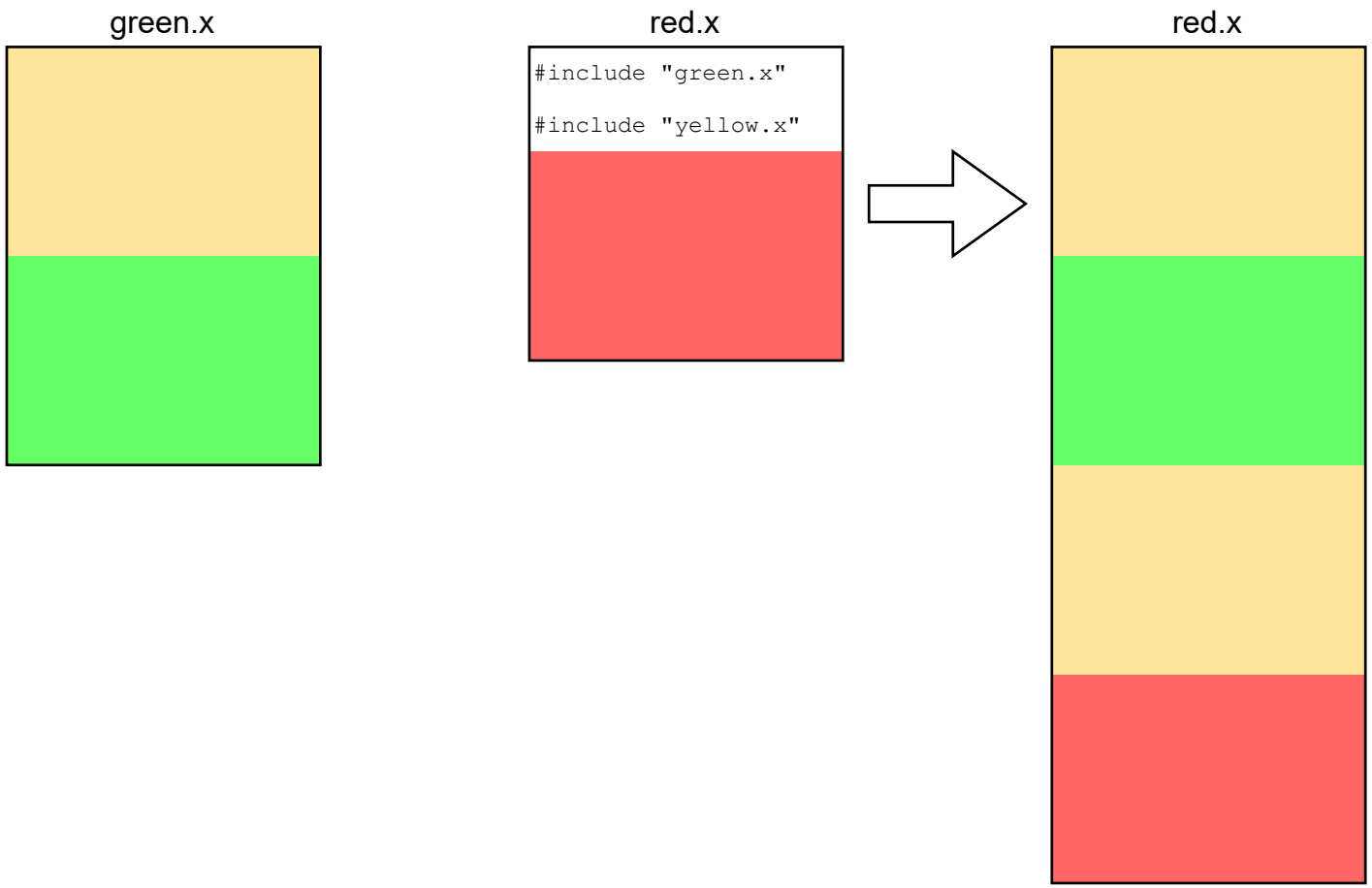


But since `#include` simply does a copy and paste, we have a problem !!

The contents of `yellow.x` will be included twice in `red.x`

Except for a few peculiar cases, this will result in an error…

The multiple inclusion problem



But since `#include` simply does a copy and paste, we have a problem !!

The contents of `yellow.x` will be included twice in `red.x`

Except for a few peculiar cases, this will result in an error…

… complaining about multiple declarations or definitions !!

The C Preprocessor - 5/5

The type of checks we saw are often required to mitigate the problem of multiple inclusion

In fact, It is so common, that there are shortcut directives for this use – `#ifndef` and `#ifdef`

- Actually, `#if !defined(X)` is equivalent to writing `#ifndef X`
- Similarly, `#if defined (X)` is equivalent to writing

The multiple inclusion problem occurs, when a piece of code gets included multiple times…

- … within the same compilation unit

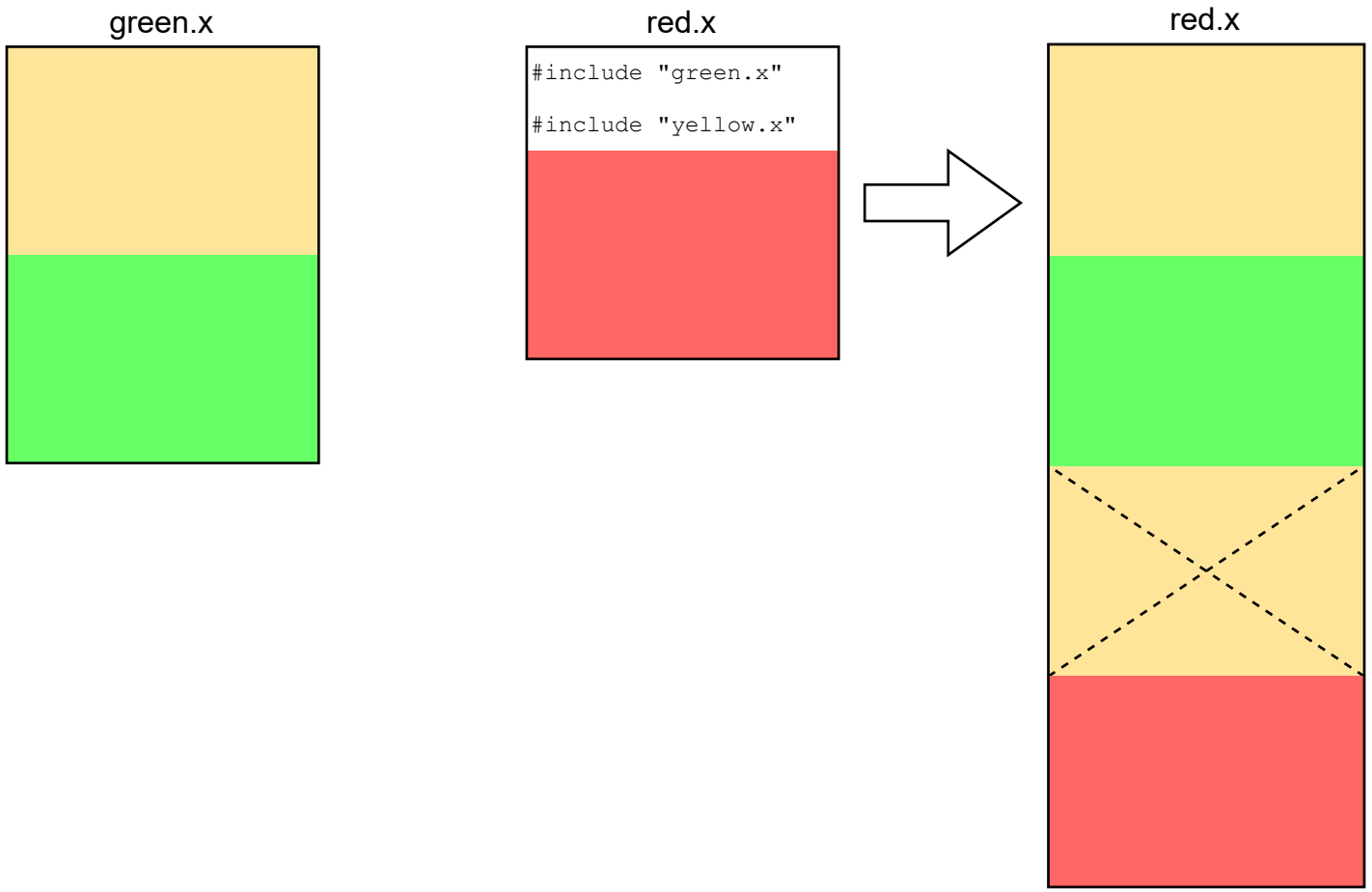
This could happen if the same file (usually a header file) gets included into multiple code files…

- … and then all these code files are compiled together to build an application

To avoid this, we define a token and use this with the `#ifndef` directive to create a guard

- The token, of course, needs to be unique across the application, and hence, requires some thinking
- A common convention is to start these tokens with an underscore

The multiple inclusion problem



In the example, we do not want the duplication of `yellow.x`'s contents

Conditional directives

yellow.x

```
#ifndef _YELLOW
#define _YELLOW

#endif
```

green.x

```
#include "yellow.x"
```

green.x

To do so, we can make a change in the `yellow.x` file

Conditional directives

yellow.x

```
#ifndef _YELLOW
#define _YELLOW

#endif
```

green.x

```
#include "yellow.x"
```

green.x

To do so, we can make a change in the `yellow.x` file

We define a token called `_YELLOW` inside an `#ifndef` directive

Conditional directives

yellow.x

```
#ifndef _YELLOW
#define _YELLOW

#endif
```

green.x

```
#include "yellow.x"
```

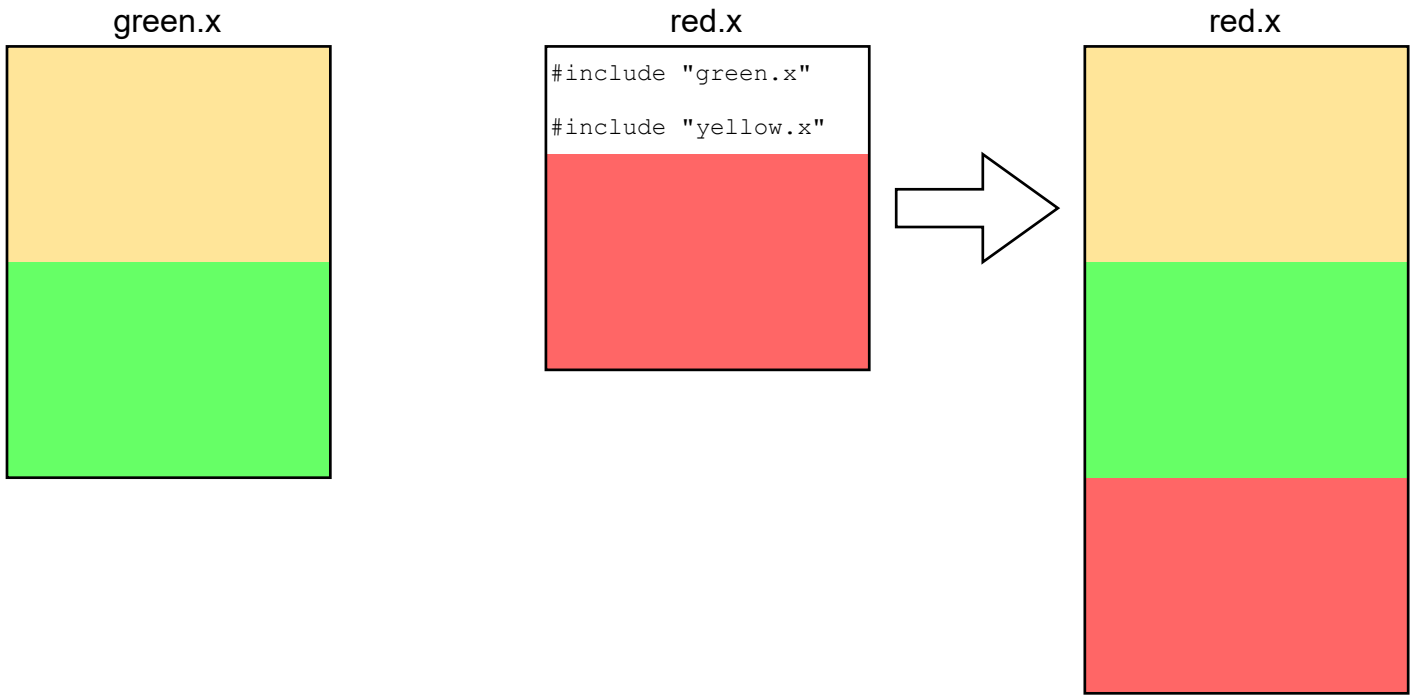
green.x

To do so, we can make a change in the `yellow.x` file

We define a token called `_YELLOW` inside an `#ifndef` directive

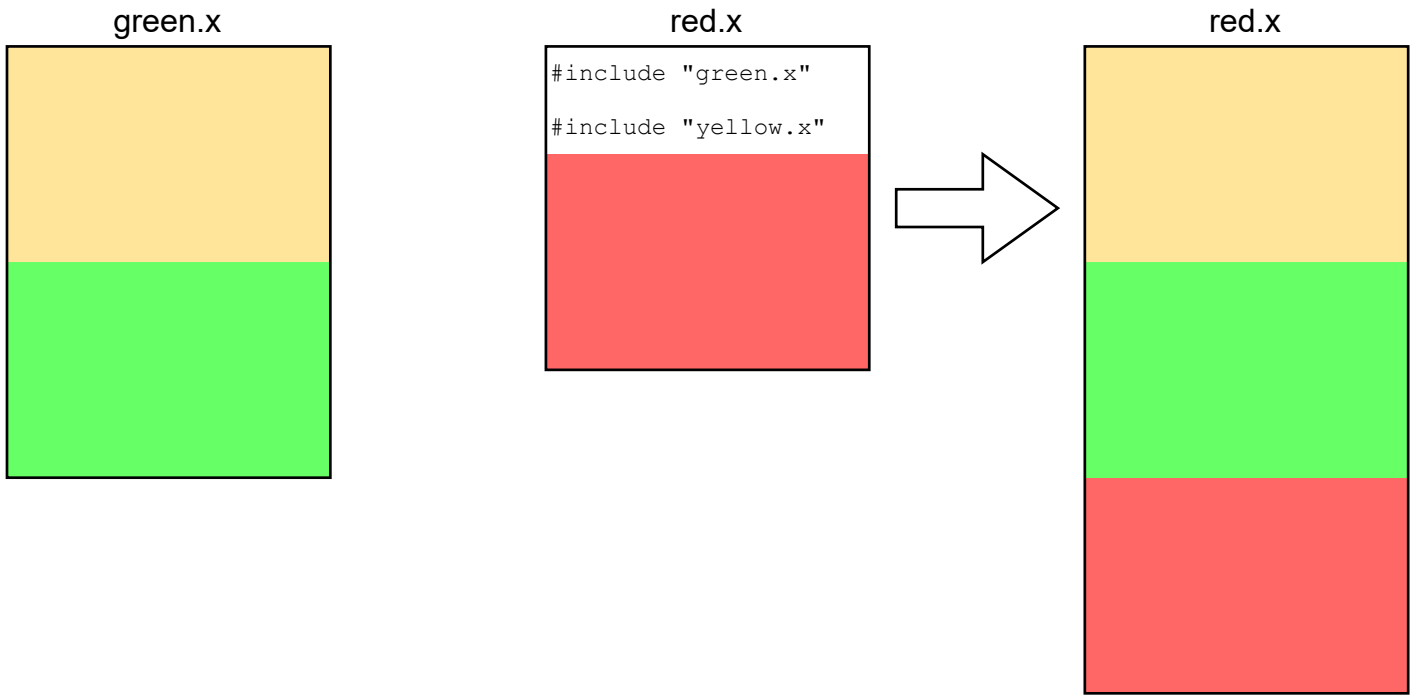
The original content of `yellow.x` gets enclosed between the `#ifndef` and `#endif` directives

Conditional directives



The `#ifndef` directive evaluates to true for the first time

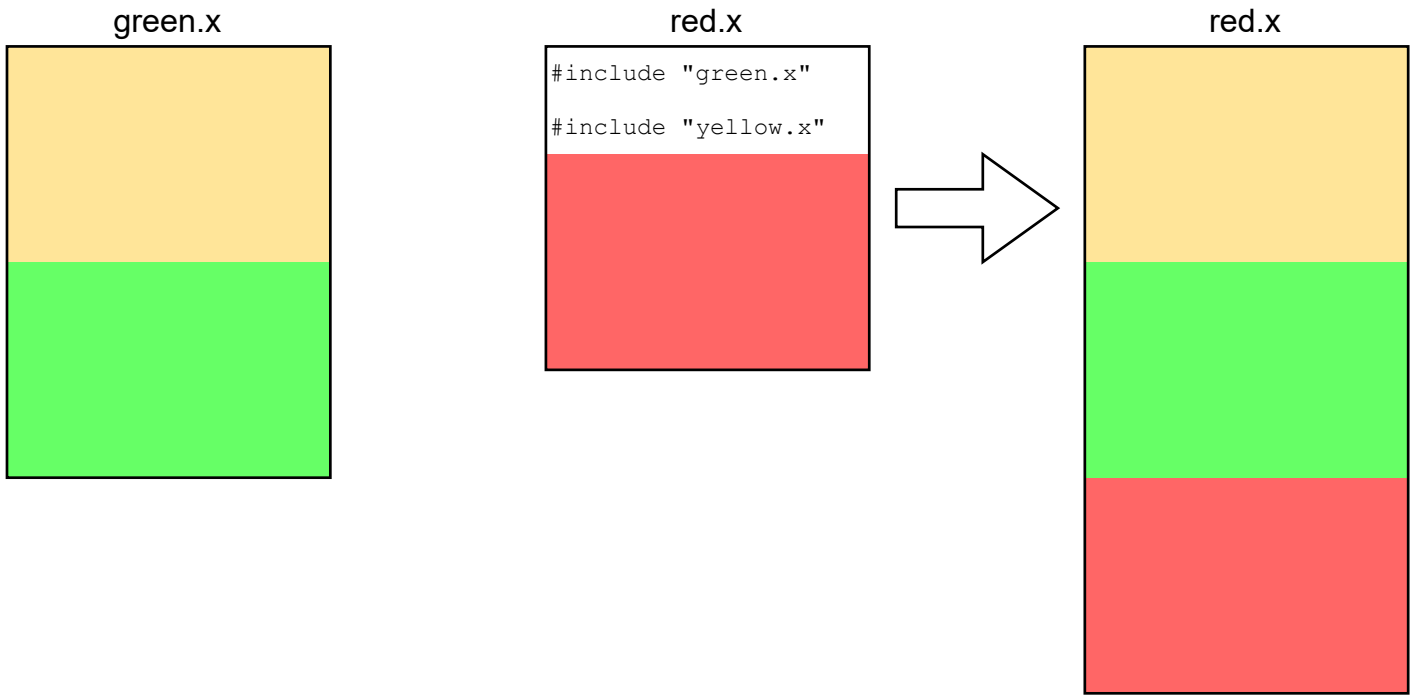
Conditional directives



The `#ifndef` directive evaluates to true for the first time

During that instance, the `_YELLOW` token gets defined

Conditional directives

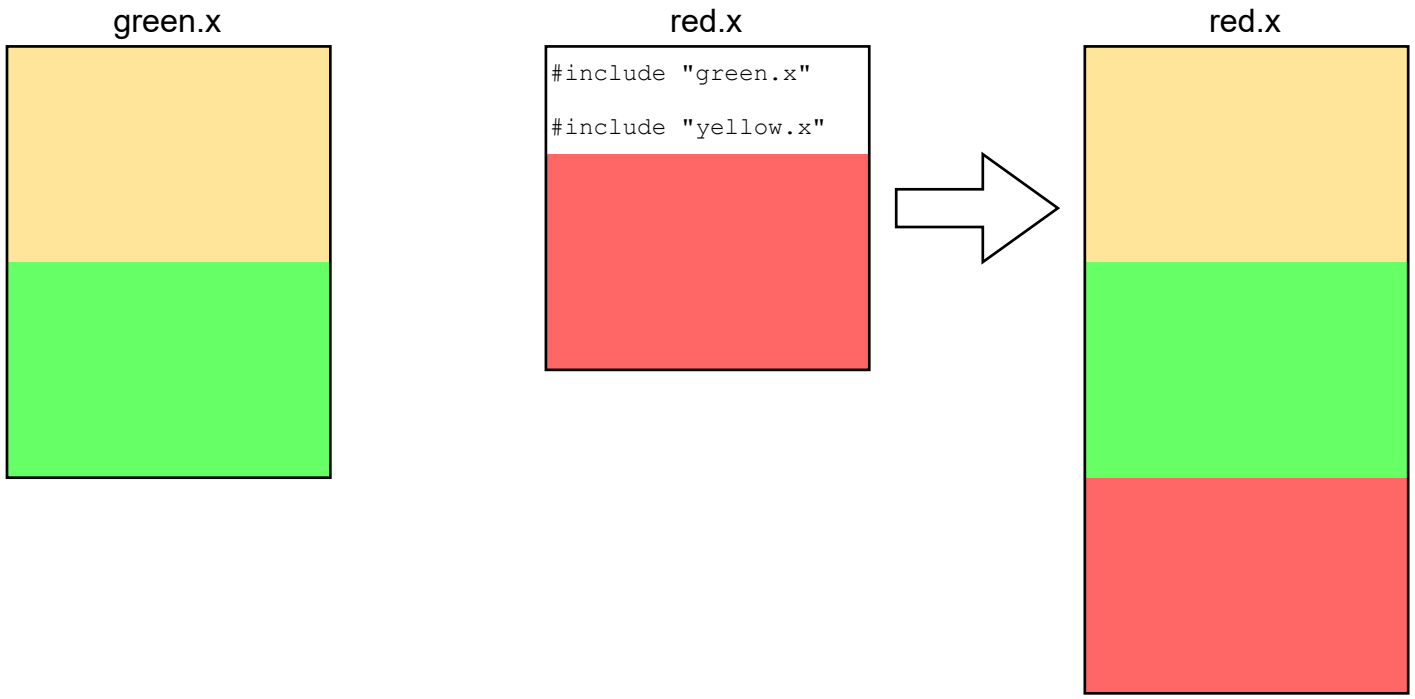


The `#ifndef` directive evaluates to true for the first time

During that instance, the `_YELLOW` token gets defined

Subsequent calls to include `yellow.x` results in `#ifndef` evaluating to false

Conditional directives



The `#ifndef` directive evaluates to true for the first time

During that instance, the `_YELLOW` token gets defined

Subsequent calls to include `yellow.x` results in `#ifndef` evaluating to false

This makes sure that the contents of `yellow.x` gets included only once in `red.x`

Detour – Compiling distributed C code

All the discussion we had till now, leaves a rather important point out

- How can we build an executable, out of code that is split across multiple files?

Detour – Compiling distributed C code

All the discussion we had till now, leaves a rather important point out

- How can we build an executable, out of code that is split across multiple files?

This is where, differentiating between compilation and linking becomes essential

Detour – Compiling distributed C code

All the discussion we had till now, leaves a rather important point out

- How can we build an executable, out of code that is split across multiple files?

This is where, differentiating between compilation and linking becomes essential

The broad steps are as follows

- We compile the individual source files with `-c` switch

Detour – Compiling distributed C code

All the discussion we had till now, leaves a rather important point out

- How can we build an executable, out of code that is split across multiple files?

This is where, differentiating between compilation and linking becomes essential

The broad steps are as follows

- We compile the individual source files with `-c` switch
- This produces a set of `.o` files, containing the compiled source code of corresponding source files

Detour – Compiling distributed C code

All the discussion we had till now, leaves a rather important point out

- How can we build an executable, out of code that is split across multiple files?

This is where, differentiating between compilation and linking becomes essential

The broad steps are as follows

- We compile the individual source files with `-c` switch
- This produces a set of `.o` files, containing the compiled source code of corresponding source files
- Then, we produce an executable by linking all the object files together

Detour – Compiling distributed C code

All the discussion we had till now, leaves a rather important point out

- How can we build an executable, out of code that is split across multiple files?

This is where, differentiating between compilation and linking becomes essential

The broad steps are as follows

- We compile the individual source files with `-c` switch
- This produces a set of `.o` files, containing the compiled source code of corresponding source files
- Then, we produce an executable by linking all the object files together
- Usually, we only compile the `.c` files (assuming that the `.h` files contain only declarations)

Detour – Compiling distributed C code

All the discussion we had till now, leaves a rather important point out

- How can we build an executable, out of code that is split across multiple files?

This is where, differentiating between compilation and linking becomes essential

The broad steps are as follows

- We compile the individual source files with `-c` switch
- This produces a set of `.o` files, containing the compiled source code of corresponding source files
- Then, we produce an executable by linking all the object files together
- Usually, we only compile the `.c` files (assuming that the `.h` files contain only declarations)

For a large application, you may be better off by creating a *make file*

- A make file contains specific instructions to compile and link code spread across multiple files
- Check the Additional Reading Slide for more details

Homework !!

Read Section 4.11 from your Reference Book

- It is just about 4 pages… do read it for getting more clarity

Play with the program called `TheSquareMacro.c` in the examples

- See if it works for different data types or not !!

Additional Reading

Making applications in C is a daunting, yet fascinating task…

- … and writing a good make file is a typical part of that exercise
- Have a look at this tutorial if you would like to know more about them https://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/

I have included a directory called `showroom` in the examples

- To create an executable out of the code, run the command `make`
- If that doesn't work, try `make -f makefile`
- You can then run the executable with the command `./showroom < sample-input.txt`

Read the code, especially, the use of `static` keyword in `ShowroomOperations.c`

- Figure out the repercussions of this
- **Hint** – there is another make file and another file called `Showroom2.c` in the directory