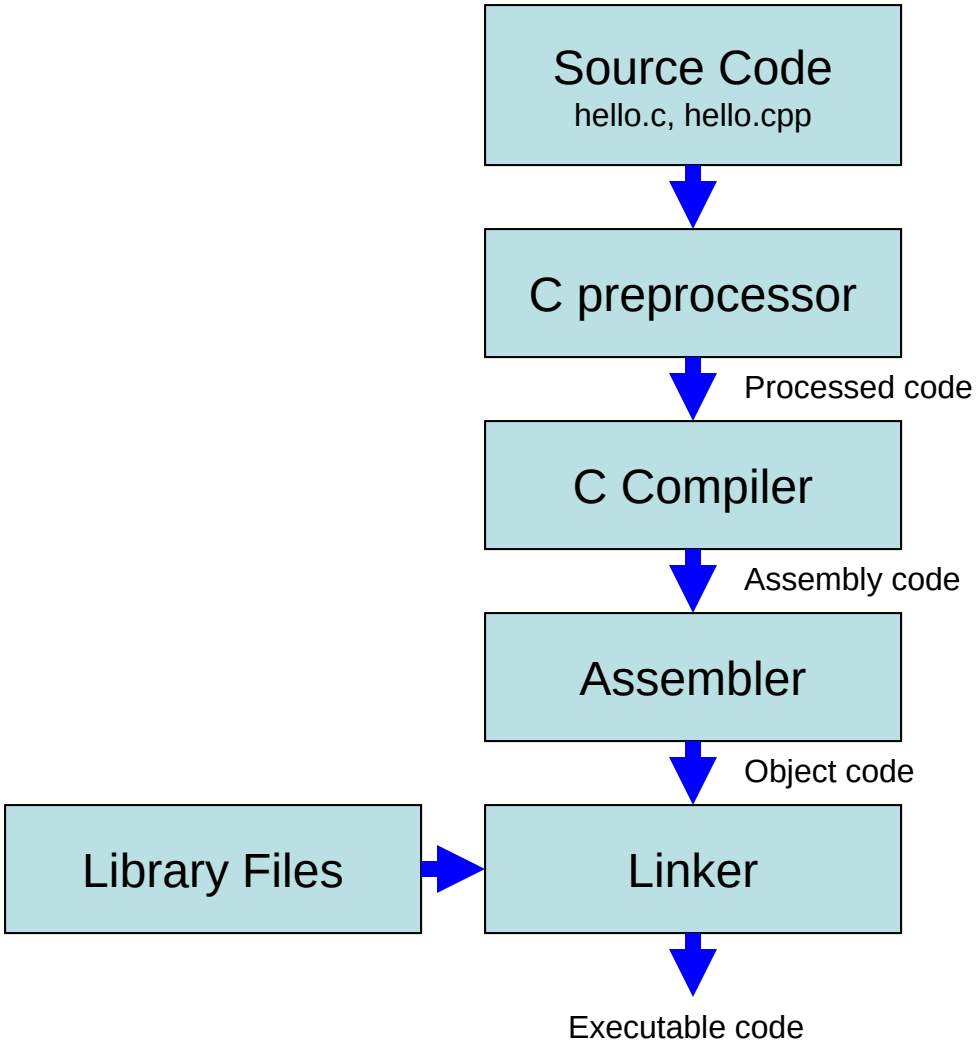
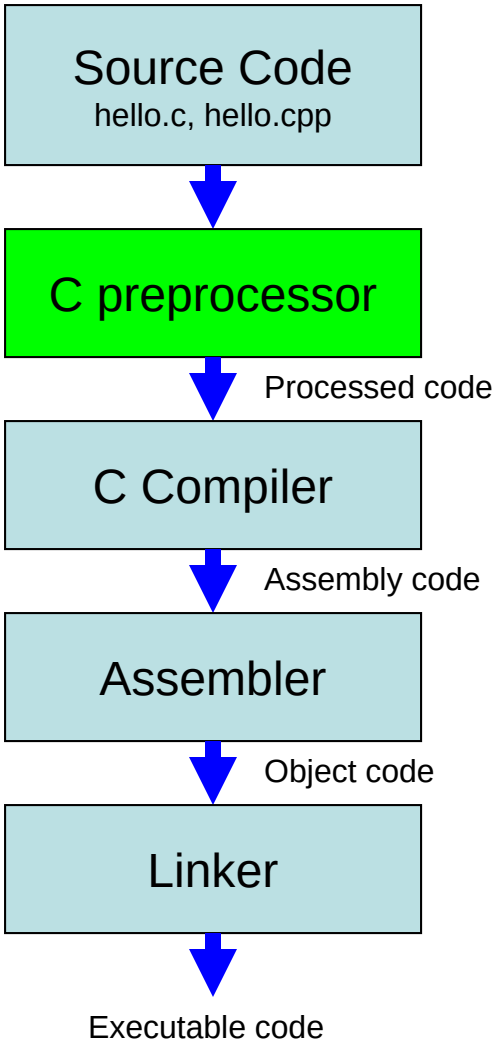
C Review

Building a C Program



C Review

The C Preprocessor



The first pass of any C compilation. It processes include-files, conditional compilation instructions and macros.

Include another file

< > means search in standard include paths

```
#include <stdio.h> // standard libraries, e.g. printf()
```

```
#include “file1.h”
```

“ ” means search in source directory

Conditional Compilation

#if, #ifdef, #ifndef, #else, #elif and #endif

```
#ifdef WIN32     // WIN32 is defined by all Windows 32 compilers
      #include <windows.h>
#else
      #include <unistd.h>
#endif
```

```
#if VERBOSE >= 2
      print("trace message");
#endif
```

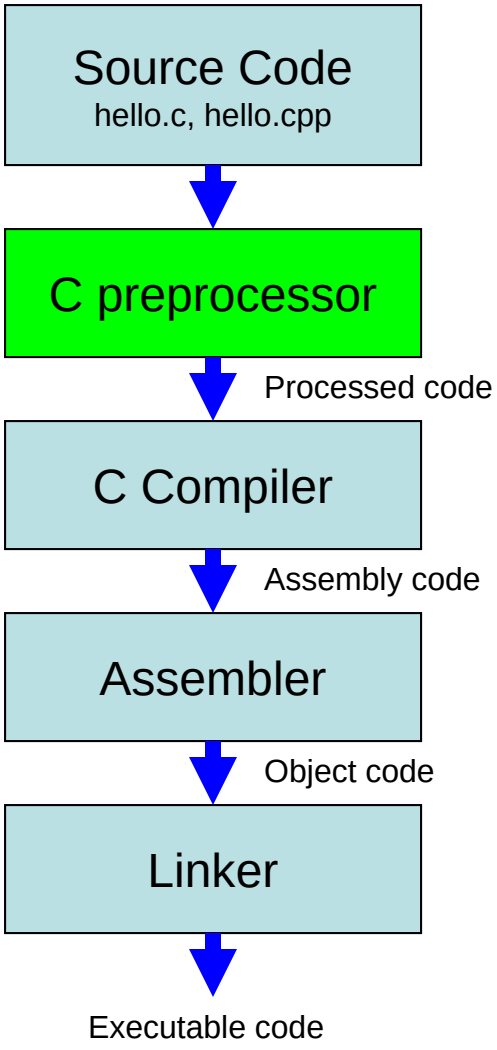
Macro Definition and Expansion

```
#define PI 3.14159
```

```
#define RADTODEG(x) ((x) * 57.29578)   // use parentheses around argument
                                       //  and expression because expansion
                                       //  can have unexpected results
```

C Review

The C Preprocessor



Include another file

#include “file1.h” ← “ ” means search in source directory

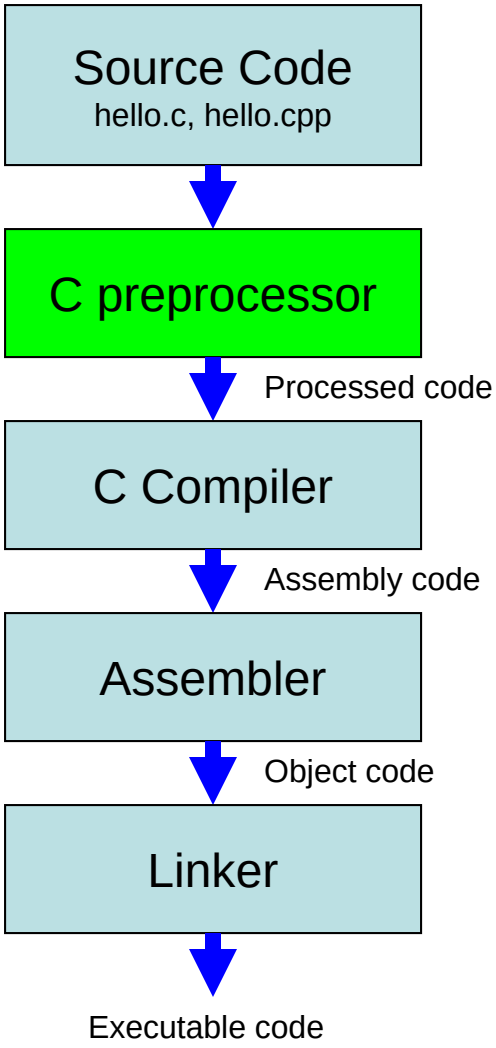
An interesting example is where I’ve processed the RESID database (a database of all documented Amino Acid post-translational modifications (PTMs)). I used Matlab to generate a C file (30,621 lines of code) that put the data into specific data structures that I included into a C program.

It was a quick way to get data into the code without having to write file I/O routines.

Now the data is in a xml format and I have a xml parser reading the file and then the data is placed in data structures.

C Review

The C Preprocessor



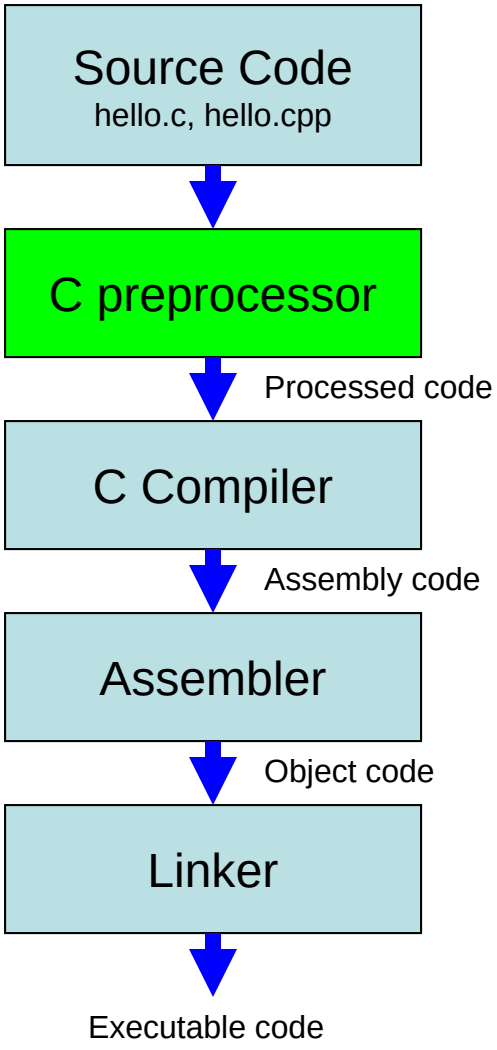
Constants

#define [identifier name] [value]
#define PI_PLUS_ONE (3.14 + 1)

In code this could be used as x = PI_PLUS_ONE * 5;
Note that without the parentheses, this would be x = 3.14 + 1 * 5;

C Review

The C Preprocessor



Conditional Compilation

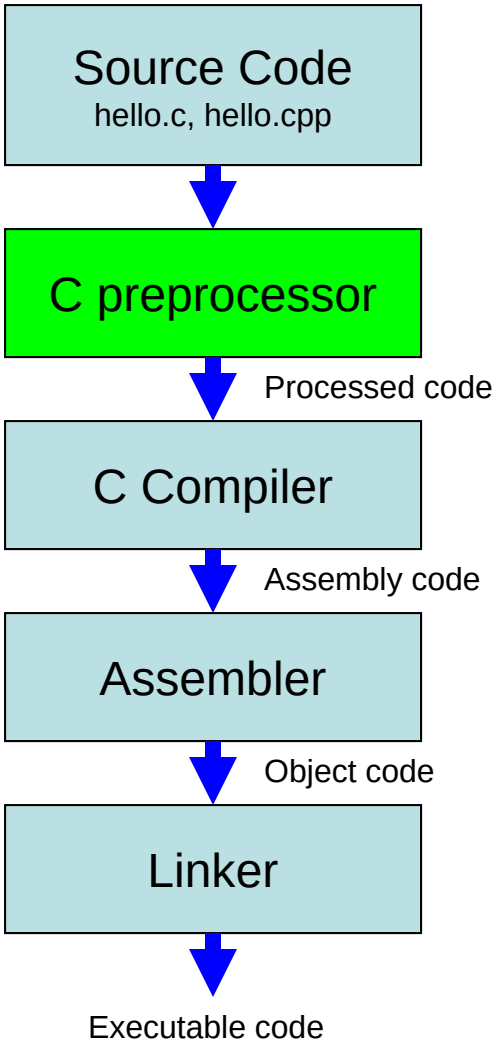
#define [indentifier name] // defines [identifier name] without giving it a value

Use #1 - Avoiding Including Files Multiple Times (idempotency)

```
#ifndef _FILE_NAME_H_
      #define _FILE_NAME_H_
      /* insert header file info, data structs, code or file here */
#endif   // #ifndef _FILE_NAME_H_
```

C Review

The C Preprocessor



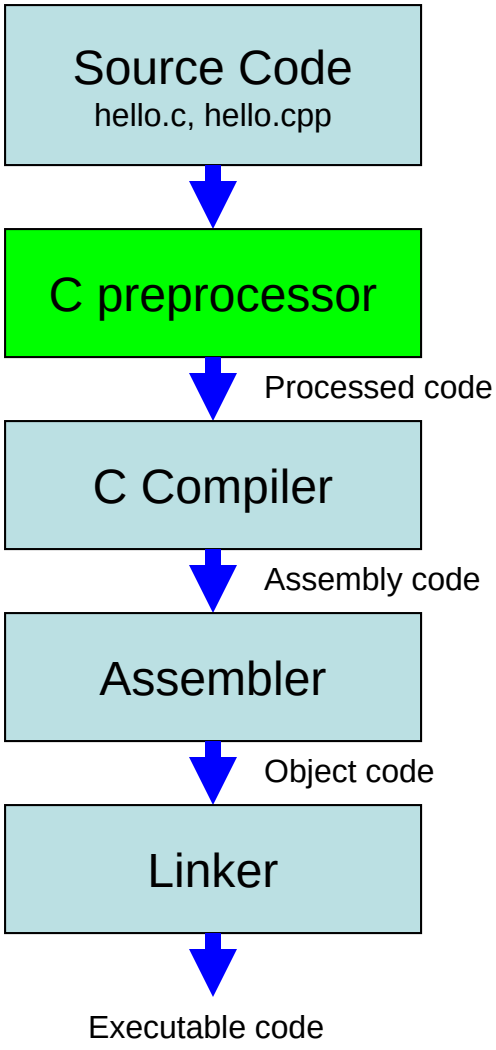
Conditional Compilation

Commenting out Code

```
#if 0
/* comment ... */
// code
/* comment */
#endif

// A useful way to comment out a block of code that
// contains multi-line //comments (which cannot be nested).
```

C Review

The C Preprocessor



Macros

#define MACRO_NAME(arg1, arg2, ...) [code to expand to]

// Macros are used to inline code in order to avoid function call overhead.

// NOTE: Remember to use a **LOT** of parentheses when using macros.

Issue #1

#define MULT(x,y) x*y

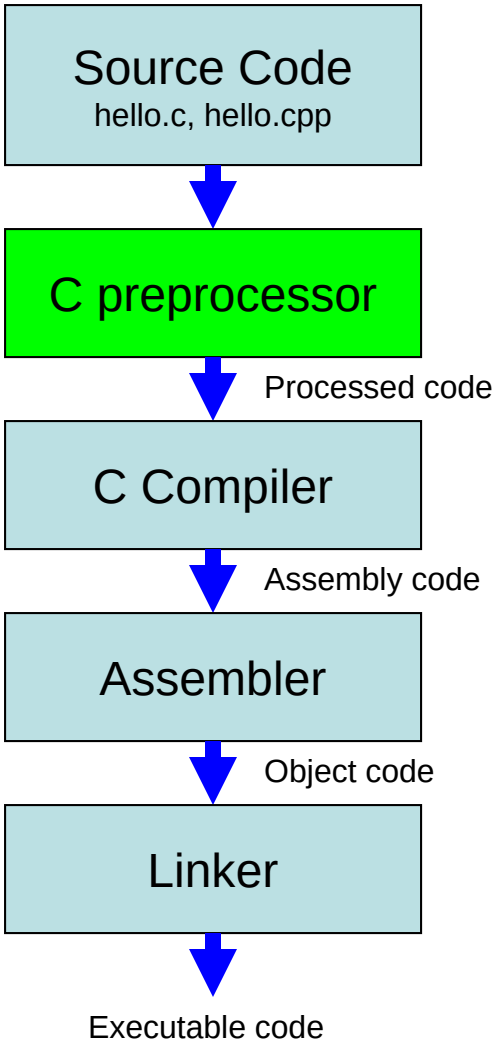
In code this could be written as int z = MULT(3 + 2, 4 + 2);
which would be expanded as z = 3 + 2 * 4 + 2;
Note: 2 * 4 will be evaluated first!

So this should be:

#define MULT(x,y) (x)*(y) // put parenthesis around the arguments
// now MULT(3 + 2, 4 + 2) will expand to (3 + 2) * (4 + 2)

C Review

The C Preprocessor



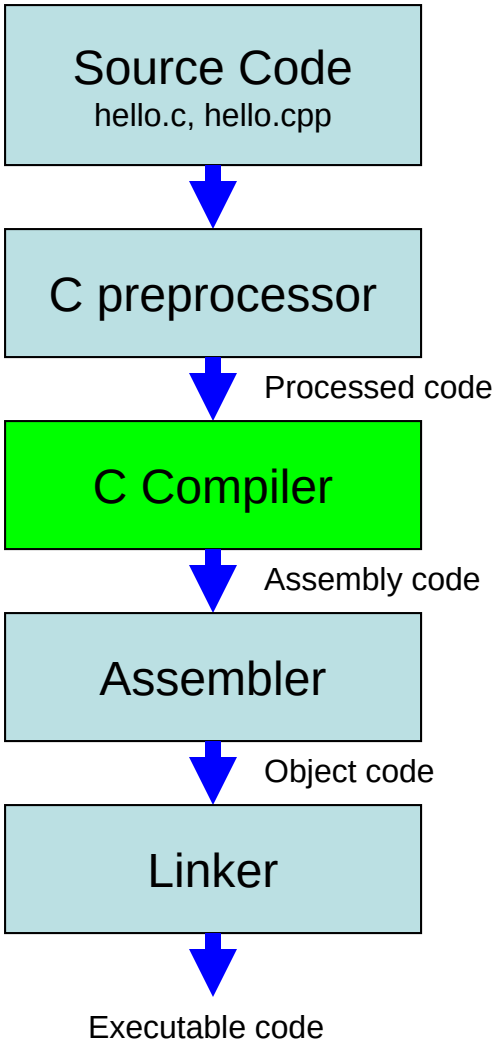
Macros

Issue #2

```
// Ok, so we put parenthesis around the arguments

#define ADD_FIVE(a)  (a)+5

In code this could be written as int x = ADD_FIVE(3) * 3;
which would be expanded as         x = 3 + 5 * 3;
// Note: x is 18, not 24, since 5*3 is evaluated first

So this should be:

#define ADD_FIVE(a) ((a) + 5)   // put parenthesis around the arguments
                                // and around the whole macro body
```

C Review

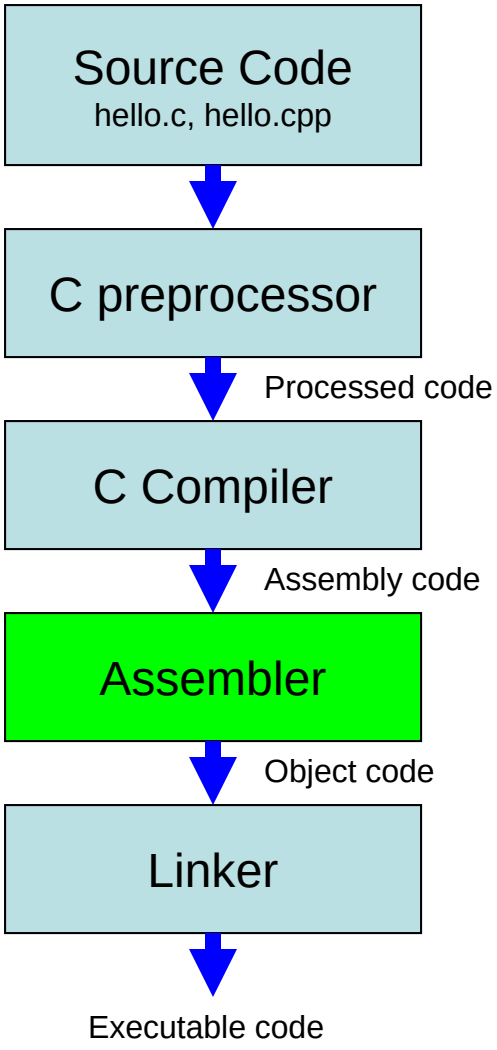
The C Compiler



Compilation is the second pass. It takes the output of the preprocessor, and the source code, and generates assembler source code.

C Review

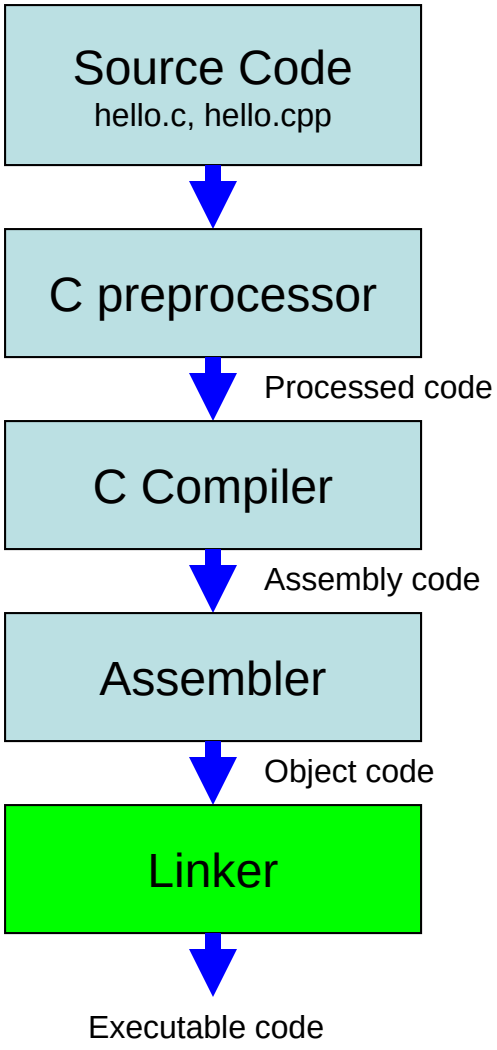
The Assembler



Assembly is the third stage of compilation. It takes the assembly source code and produces an assembly listing with offsets. The assembler output is stored in an object file.

C Review

The Linker



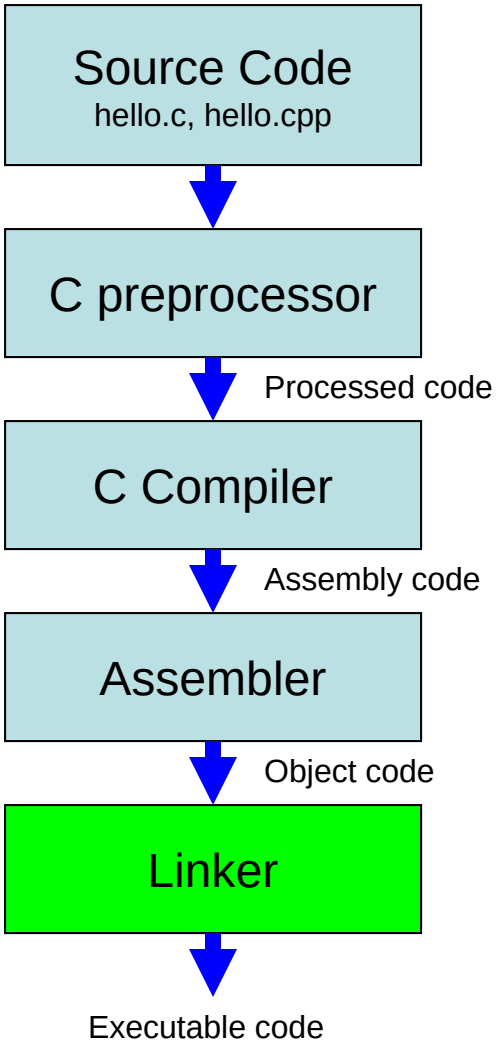
Linking is the final stage of compilation. It takes one or more object files or libraries as input and combines them to produce a single (usually executable) file. In doing so, it resolves references to external symbols, assigns final addresses to procedures/functions and variables, and revises code and data to reflect new addresses (a process called relocation).

Sections that are common to all executable formats

.text	This section holds the "text," or executable instructions, of a program.
.bss	This section holds uninitialized data that contribute to the program's memory image. By definition, the system initializes the data with zeros when the program begins to run.
.data	Contains the initialized global and static variables and their values. It is usually the largest part of the executable. It usually has READ/WRITE permissions.
.rodata	These sections hold **read-only data** that typically contribute to a non-writable segment in the process image.

C Review

The Linker Settings for the **Nios II** IDE



The linker settings in the Nios II IDE allow the placement of the following sections and memory blocks in different memory locations

.text This section holds the "text," or executable instructions, of a program.

.rodata These sections hold **read-only data** that typically contribute to a non-writable segment in the process image.

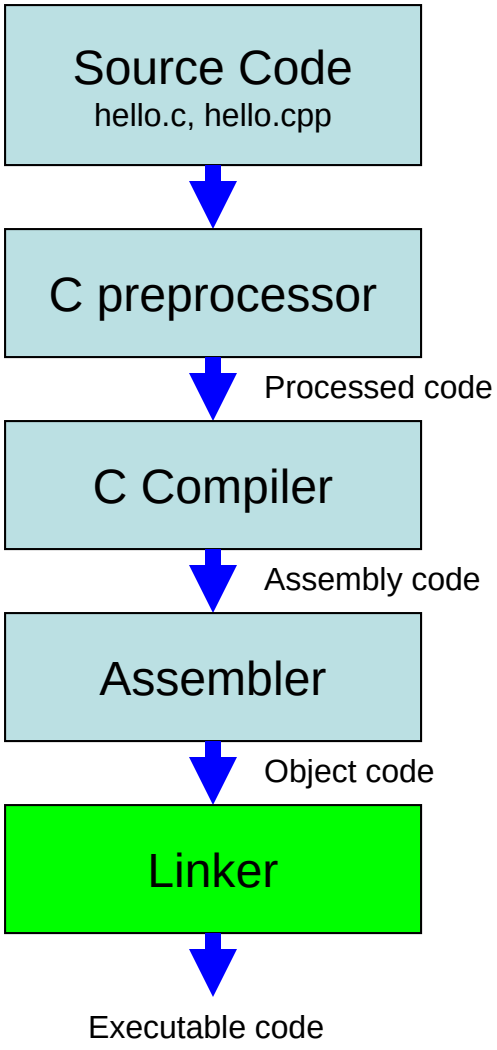
.rwdata The same as **.data**
Contains the initialized global and static variables and their values. It is usually the largest part of the executable. It usually has READ/WRITE permissions.

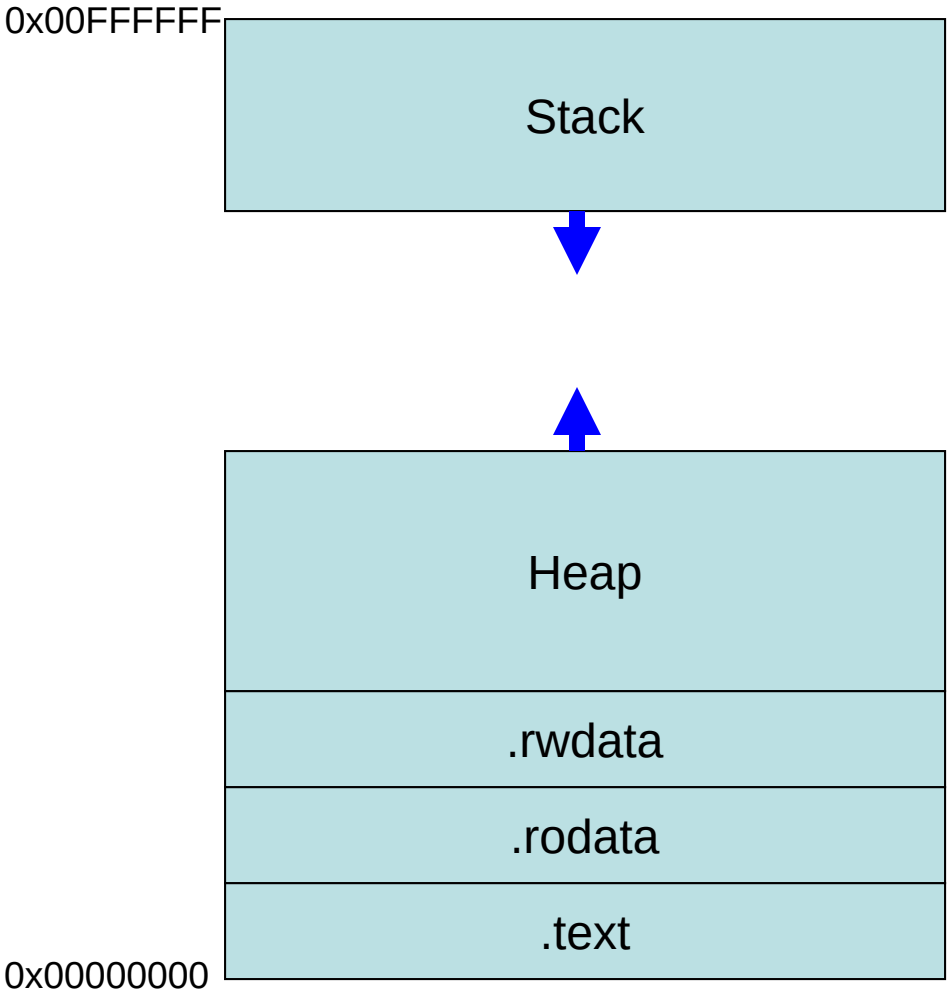
Heap Location of dynamic memory allocation

Stack Location of stack frames for function calls

C Review

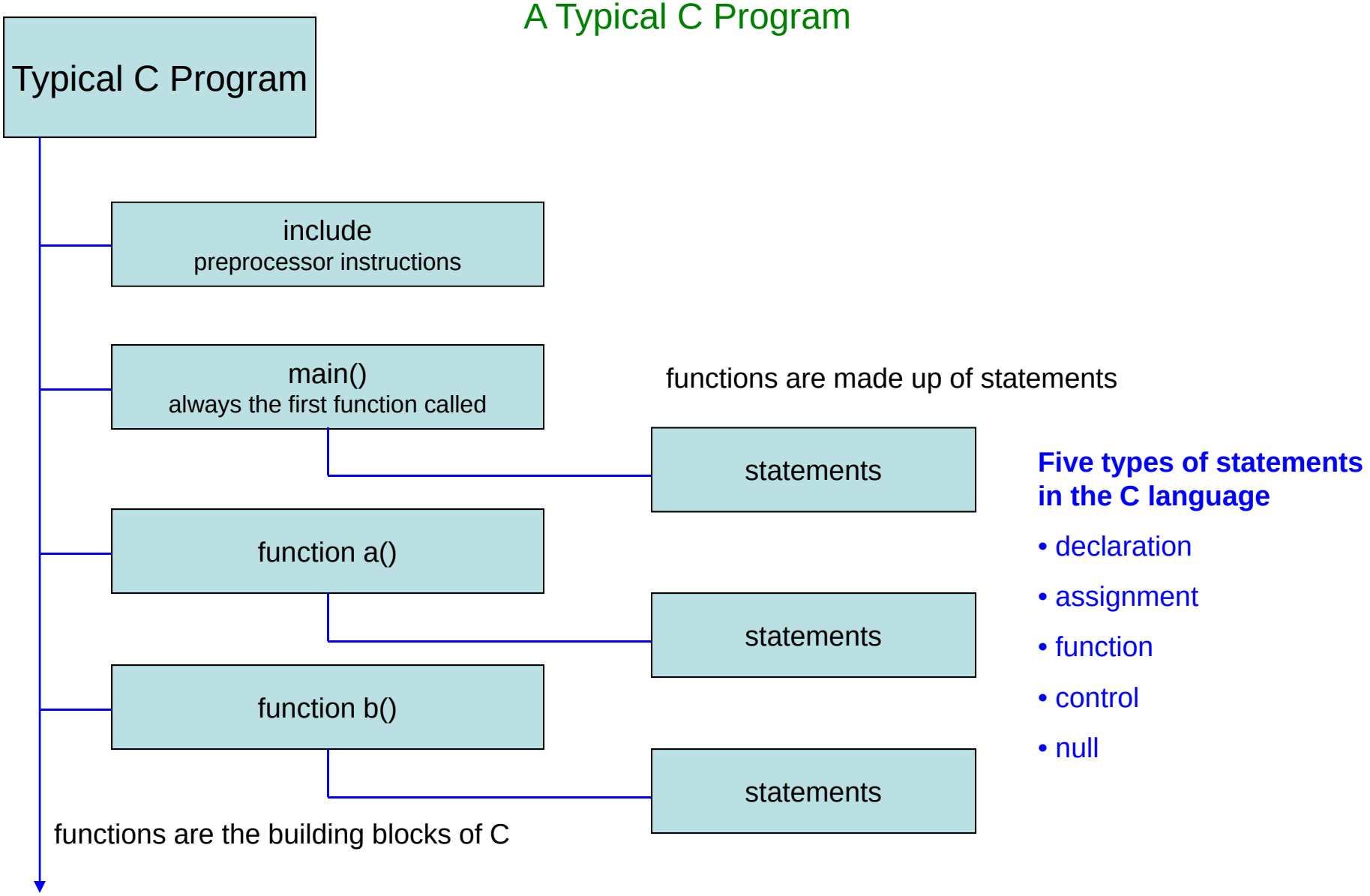
Nios II Memory Map





C Review

A Typical C Program



Five types of statements in the C language

- declaration
- assignment
- function
- control
- null

Why can’t programmers tell Halloween from Christmas Day?

Why can’t programmers tell Halloween from Christmas Day?

because

OCT 31 = DEC 25

i.e.

$31_8 = 3x8^1 + 1x8^0 = 25_{10}$

What does this C program generate?

```
#include <stdio.h>
main(t,_,a)char *a;{return!0<t?t<3?main(-79,-13,a+main(-87,1-_, main(-
86,0,a+1)+a)):1,t<_?main(t+1,_,a):3,main(-94,-27+t,a)&&t==2?_<13?
main(2,_+1,"%s %d %d\n"):9:16:t<0?t<-72?main(_,t,
"@n'+,#'/*{}w+/w#cdnr/+,{}r/*de}+,/*{*+,/w{%+,/w#q#n+,/#{l,+,/n{n+,/+#n+,/#\
;#q#n+,/+k#;*+,/'r :'d*'3,}{w+K w'K:'+}e#';dq#'l \
q#'+d'K#!/+k#;q#'r}eKK#}w'r}eKK{nl]'/#;#q#n')(){#}w'){){nl]'/+#n';d}rw' i;# \
){nl]!/n{n#'; r{#w'r nc{nl]'/#{l,+'K {rw' iK{;[{nl]'/w#q#n'wk nw' \
iwk{KK{nl]!/w{%'l##w#' i; :{nl]'/*{q#'ld;r'}{nlwb!/*de}'c \ ;;{nl'-
{}rw]'/+,}##'*}#nc,',#nw]'/+kd'+e}+;#'rdq#w! nr'/ ') }+}{rl#'{n' ')# \ }'+}##(!!/'') :t<-
50?_==*a?putchar(31[a]):main(-65,_,a+1):main((*a=='/')+t,_,a+1)
:0<t?main(2,2,"%s"):*a=='/'||main(0,main(-61,*a, "!ek;dc i@bK'(q)-
[w]*%n+r3#l,{}:\nuwloca-O;m .vpbks,fxntdCeghiry"),a+1);}
```

It is a legal C program that when compiled and run will generate the 12 verses of *The 12 Days of Christmas*.

A winning entry from the International Obfuscated C Code competition written by Ian Phillipps in 1988.

A great example of what NOT to do when trying to create readable and maintainable code.

A simple C program

```
#include <stdio.h>
void main(void){
    printf(“Microcontrollers run the world!\n”);
}
```

Debugging is twice as hard as writing code in the first place. Therefore, if you write code as cleverly as possible, you are, by definition, not smart enough to debug it.

--Brian Kernighan

Brian Kernighan is a computer scientist who worked at Bell Labs alongside Unix creators Ken Thompson and Dennis Ritchie and contributed greatly to Unix and its school of thought. He is also coauthor of the AWK and AMPL programming languages. The 'K' of K&R C and the 'K' in AWK both stand for 'Kernighan'.

Kernighan's name became widely known through co-authorship of the first book on the C programming language with Dennis Ritchie. Kernighan has said that he had no part in the design of the C language ("it's entirely Dennis Ritchie's work"). He authored many Unix programs, including ditroff, and cron for Version 7 Unix.

Some interesting Facts on Code Development and Debugging.

Industry analysts report that only 47 of 220 work days are actually spent on developing planned features of new applications.

In fact, more than 30% of developer time is spent on testing and bug repairs.

Industry analysts estimate that a defect detected in deployment is a minimum of 100 times more costly than a defect detected in development.

Information from www.coverity.com

C Review

Storage Classes

auto (default) : uninitialized, stored on stack, created when function is entered, can be initialized.

static : initialized to zero (.bss or .rwdata), not stored on the stack, will keep its value when the function returns and enters again.

register : int, long, short, char. The compiler will attempt to store it in a computer register. Fast access. Useful for tight loops. Note: You get better code if the compiler chooses which variable should be in the registers.

static global : can only be accessed from within the file it was defined. An extern variable from another file cannot access it.

const : must be initialized. The value cannot be changed by the program.

volatile : the value of the variable can change by causes outside of the function. **It is important to use this storage class in embedded systems when using optimizing compilers.**

C Review

Problems in embedded systems that can occur by ***not*** using volatile

To declare a variable volatile, include the keyword volatile before or after the data type in the variable definition.

volatile int foo;
int volatile foo;

Memory-mapped peripheral registers

```
uint16 * pReg = (uint16 *) 0x1234; // an 16-bit status register that is memory mapped to address 0x1234
// poll the status register until it becomes non-zero
while (*pReg == 0) { }
```

An *optimizing compiler* will generate the following assembly code:

```
mov ptr, #0x1234
mov a, @ptr
loop:
        bz loop     // having already read the variable's value into the accumulator (on the second line of assembly), there is no need to reread it, since the value will always be the same. Thus, in the third line, we end up with an infinite loop.
```

uint16 volatile * pReg = (uint16 volatile *) 0x1234; // the correct declaration

```
mov ptr, #0x1234
loop:
    mov a, @ptr          // generated code now checks the value
    bz loop
```

C Review

Problems in embedded systems that can occur by not using volatile

Interrupt service routines

```
int waitv = FALSE;

void main() {
    ...
    while (! waitv) {
        // Wait
    }
    ...
}

interrupt void isr1(void) {
    ...
    if (input == 'w') {
        waitv = TRUE;
    }
    ...
}
```

As far as the optimizing compiler is concerned, the expression !waitv is always true, and, therefore, you will never exit the while loop

The solution is to declare the variable waitv to be *volatile*.

C Review

Problems in embedded systems that can occur by not using volatile

Multi-threaded applications

```
int cntr;

void task1(void) {
    cntr = 0;
    while (cntr == 0) {
        sleep(1);
    }
    ...
}

void task2(void) {
    ...
    cntr++;
    sleep(10);
    ...
}
```

As far as the optimizing compiler is concerned, cntr is always zero, and, therefore, task1 will always be sleeping.

The solution is to declare all shared global variables to be volatile.

volatile examples from http://www.netrino.com/node/80

C Review

Problems in embedded systems that can occur by not using volatile

Summary

Embedded system need to use the volatile storage type for

1. Memory mapped registers
2. Globally shared variables (multi-threaded apps)

Otherwise you run the risk of an optimizing compiling thinking that the value is static and eliminating the code.