# C Programming Language Review and Dissection I

Lecture 3

# Today

High-level review of C concepts

*coupled with . . .*

In-depth examination of *how they are implemented* in assembly language

Reading Assignment

- MCPM Chapter 1 and Chapter 2 (Section 2.1) *Memory Mapping*
- Review P&P Chapters 14, 15, 16

# C: A High-Level Language

## Gives symbolic names to values

- don’t need to know which register or memory location

## Provides abstraction of underlying hardware

- operations do not depend on instruction set
- example: can write “a = b * c”, even if
  CPU doesn’t have a multiply instruction

## Provides expressiveness

- use meaningful symbols that convey meaning
- simple expressions for common control patterns (if-then-else)

## Enhances code readability

## Safeguards against bugs

- can enforce rules or conditions at compile-time or run-time

# A C Code "Project"

- You will use an "Integrated Development Environment" (IDE) to develop, compile, load, and debug your code.
- Your entire code package is called a *project*. Often you create several files to spilt the functionality:
  - Several C files
  - Several include (.h) files
  - Maybe some assembly language (.a30) files
  - Maybe some assembly language include (.inc) files

- A lab, like "Lab7", will be your project. You may have three .c, three .h, one .a30, and one .inc files.
- More will be discussed in a later set of notes.

# Compiling a C Program

Entire mechanism is usually called the "compiler"

Preprocessor

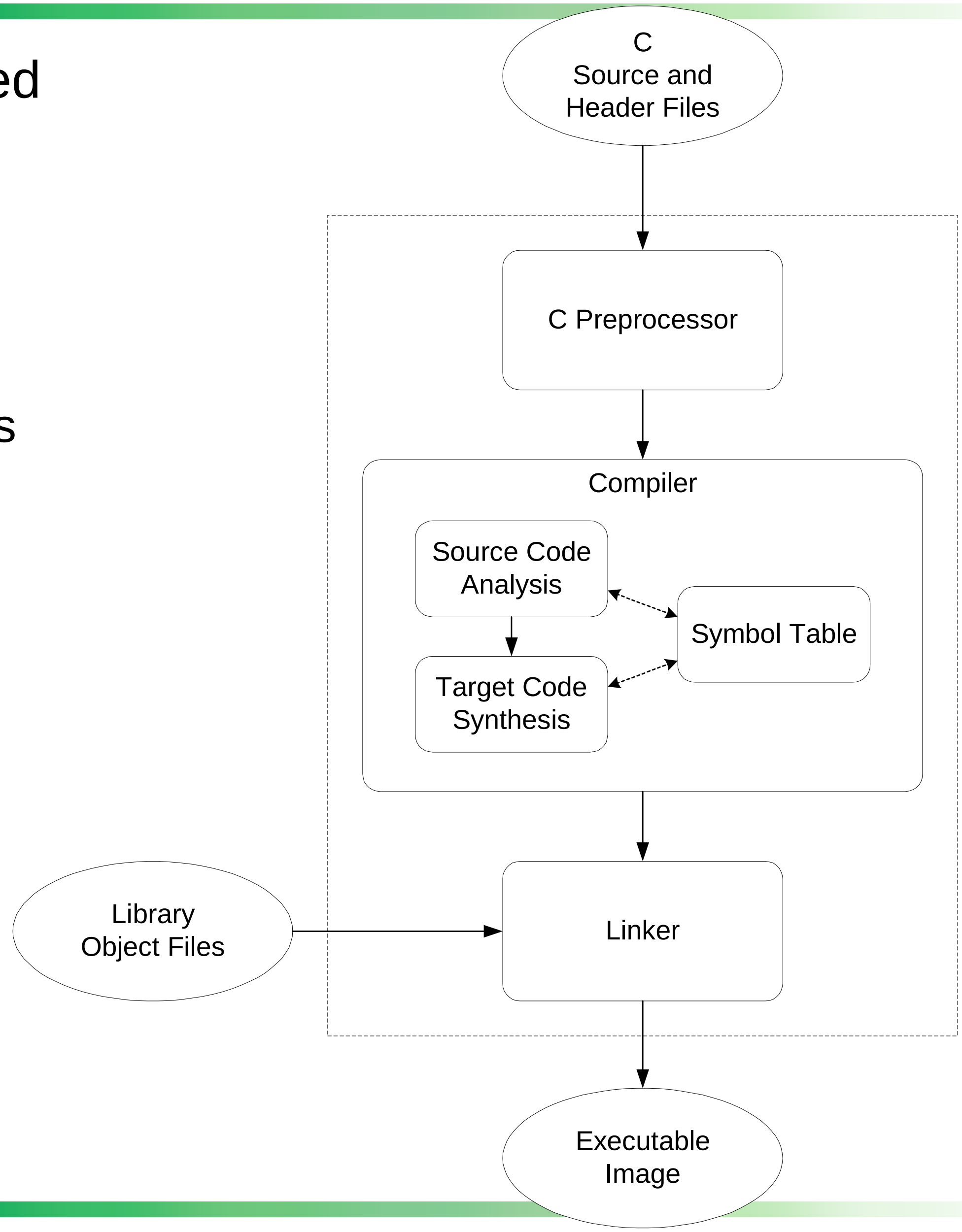
- macro substitution
- conditional compilation
- "source-level" transformations
  - output is still C

Compiler

- generates object file
  - machine instructions

Linker

- combine object files (including libraries) into executable image



# Compiler

## Source Code Analysis

- "front end"
- parses programs to identify its pieces
  - variables, expressions, statements, functions, etc.
- depends on language (not on target machine)

## Code Generation

- "back end"
- generates machine code from analyzed source
- may optimize machine code to make it run more efficiently
- very dependent on target machine

## Symbol Table

- map between symbolic names and items
- like assembler, but more kinds of information

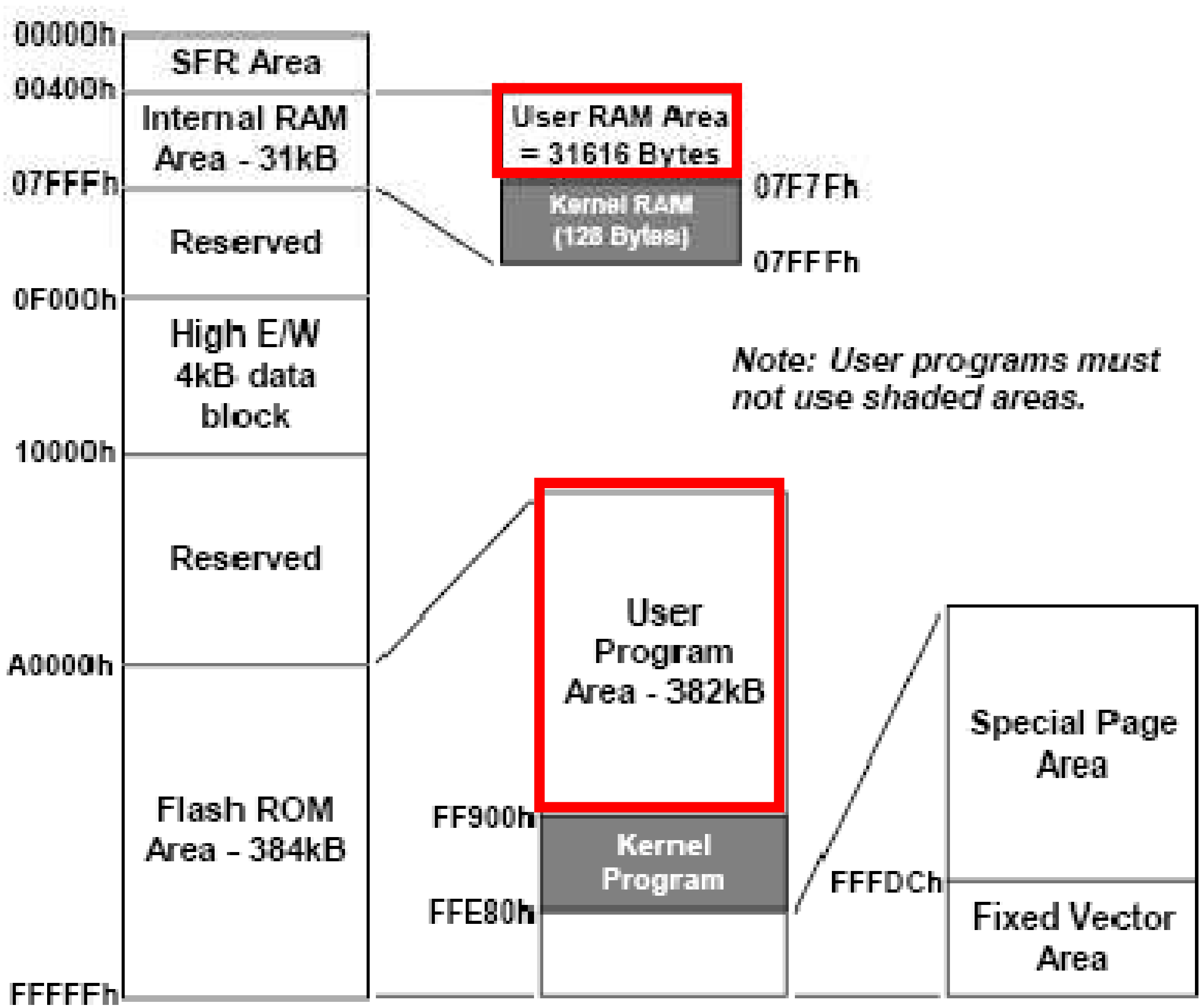
# Useful Tip

Configure Project Editor to tell compiler to generate assembly code for examination with debug information

- Option Browser -> select **CFLAGS**, select **Mod…**, select Category **et cetera** -> check **–dsource**

Also, do not use spaces in file names or directories.

# Remember the Memory Map for Our MCU



# Classifying Data

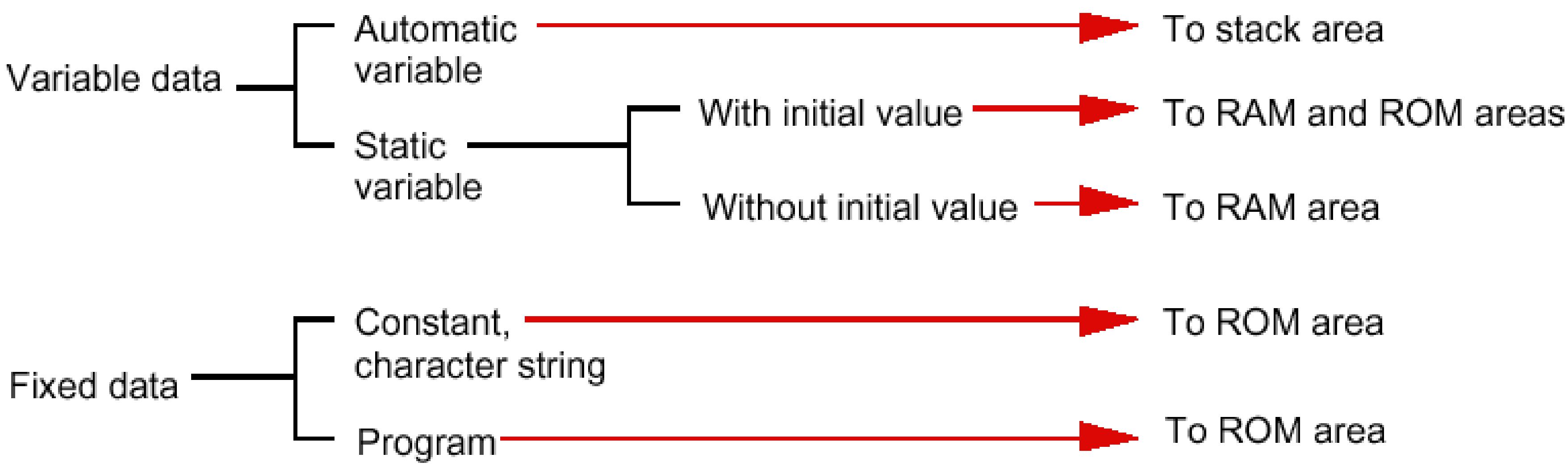



**Figure 2.1.1 Types of data and code generated by NC30 and their mapped areas**

# Section Names and Contents

**Table 2.1.1 Sections types Managed by NC30**

| Section base name | Content |
|---|---|
| data | Contains static variables with initial values. |
| bss | Contains static variables without initial values. |
| rom | Contains character strings and constants. |
| program | Contains programs. |
| vector | Variable vector area (compiler does not generate) |
| fvector | Fixed vector area (compiler does not generate) |
| stack | Stack area (compiler does not generate) |
| heap | Heap area (compiler does not generate) |

*"Block started by symbol"*

# Example of Sections

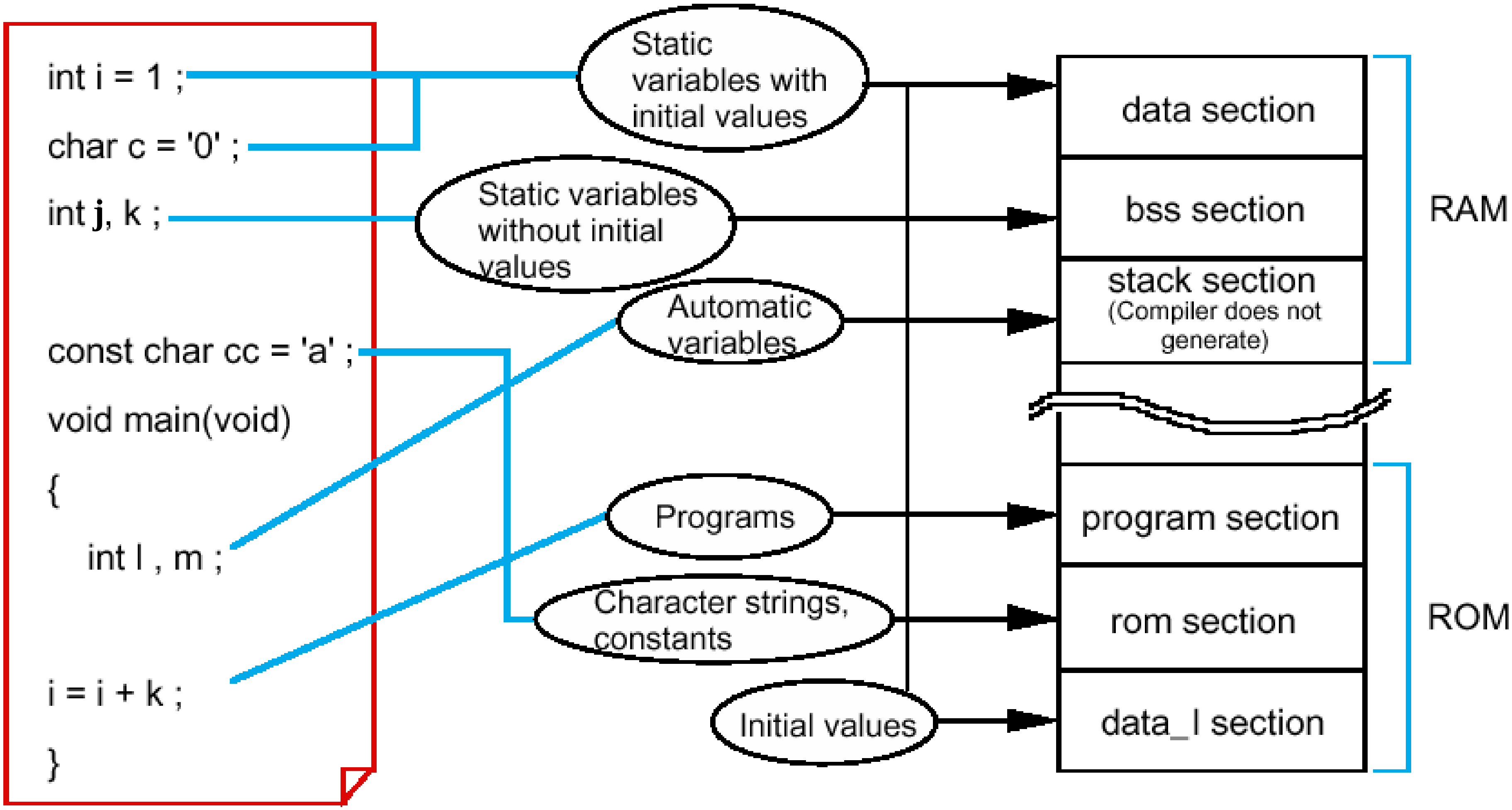



**Figure 2.1.3 Mapping data into sections by type**

# Section Sizes and Locations

Map Viewer - Shows memory map with sections and symbols

| Address(size) | Section |
|---|---|
| 000000(000400) >> | |
| 000400(000002) | [D] data_NE |
| 000402(000008) | [D] ustack |
| 00040a(000375) | |
| 00077f(000001) | [D] istack |
| 000780(0f9880) | |
| 0fa000(000002) | [R] data_NEI |
| 0fa002(0000b4) | [C] interrupt |
| 0fa0b6(0012e2) | [C] program |
| 0fb398(004468) | |
| 0ff800(0000c0) | [C] vector |
| 0ff8c0(00071c) | |
| 0fffdc(000024) | [C] fvector |
| 0fffff | |

Builder gives summary

```
DATA      0000011(0000BH) Byte(s)
ROMDATA   0000002(00002H) Byte(s)
CODE      0005242(0147AH) Byte(s)
********* Finish...
```

| Address(size) | Section | Label: |
|---|---|---|
| 000000(000400) | | |
| 000400(000002) | [D] data_NE | [G] 000400:__SB__<br>[G] 000400:_a |
| 000402(000008) | [D] ustack | |
| 00040a(000375) | | |
| 00077f(000001) | [D] istack | |
| 000780(0f9880) | | |
| 0fa000(000002) | [R] data_NEI | |
| 0fa002(0000b4) | [C] interrupt | [G] 0fa002:start<br>[G] 0fa0ae:$exit<br>[G] 0fa0ae:_exit |
| 0fa0b6(0012e2) | [C] program | [G] 0fa0b6:_init_switches<br>[G] 0fa0c4:_init_LEDs<br>[G] 0fa0de:_main<br>[G] 0fa21e:__f8add<br>[G] 0facf2:__f8mul<br>[G] 0fafd0:__f8toi4U<br>[G] 0faffa:$_ftol<br>[G] 0fb192:__i4Utof8<br>[G] 0fb1b8:$_ltof<br>[G] 0fb2c0:$_f8ltor<br>[G] 0fb332:$_f8rtol |
| 0fb398(004468) | | |
| 0ff800(0000c0) | [C] vector | |
| 0ff8c0(00071c) | | |
| 0fffdc(000024) | [C] fvector | |

# Allocating Space for Variables

Static data section

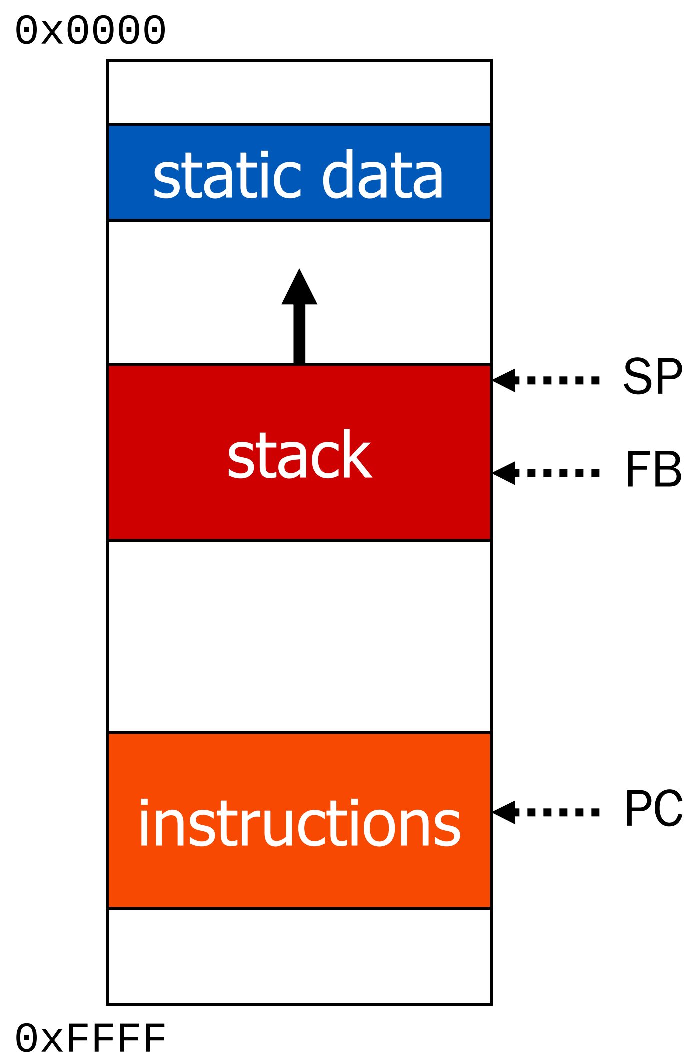
- All static variables stored here (including global variables)
- There is a fixed (absolute) address

Run-time stack

- Used for automatic variables
- SP and FB point to *storage area* (frame, activation record) at top of stack
- New storage area for each block (goes away when block exited)

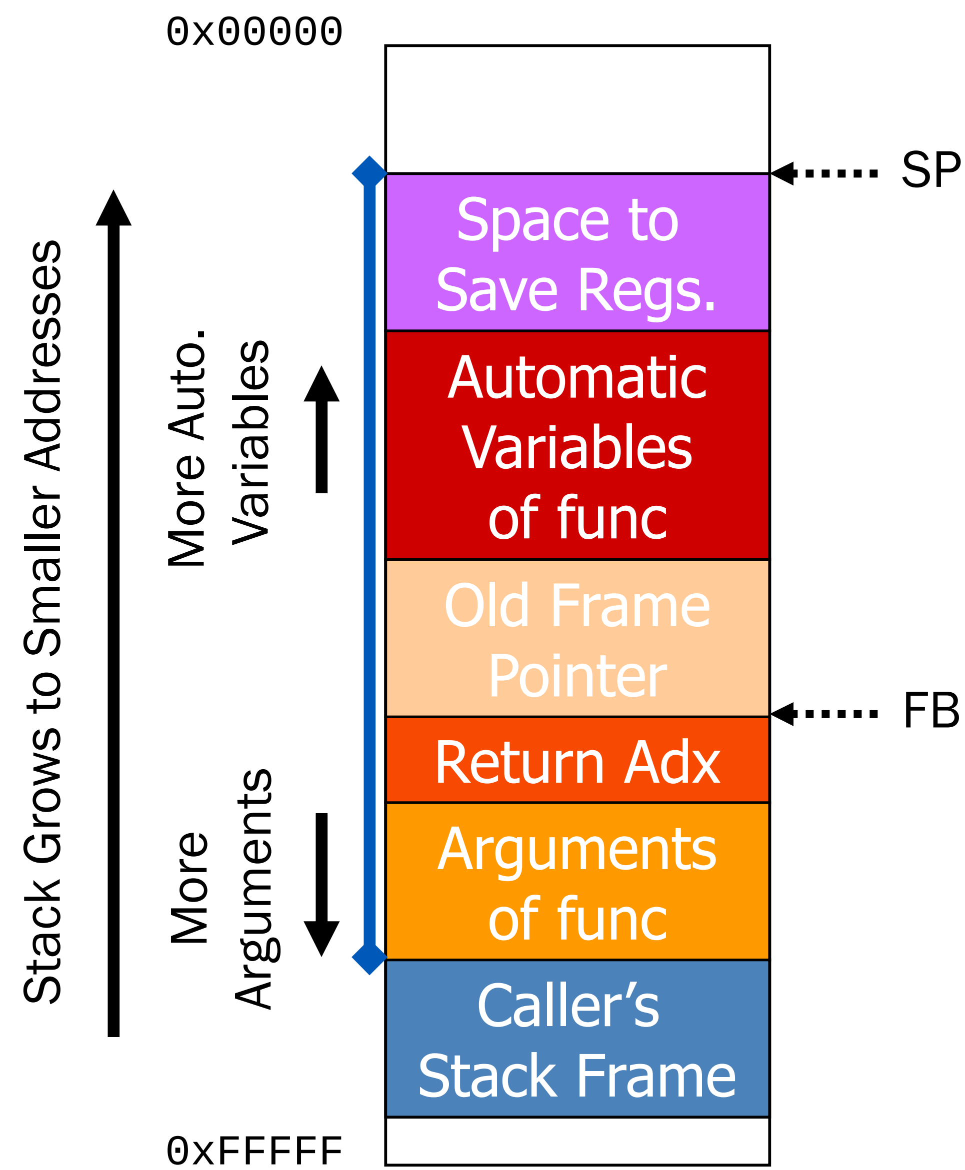
Examples

- Global: `sub.w _inGlobal,R1`
- Local: `mov.w -2[FB],R0`
  - Offset = distance from beginning of storage area



# Activation Record / Stack Frame

- Read Patt & Patel Chapter 14 for a thorough explanation of concepts
- See Section 2.4 of MCPM for more implementation details
- See Section 1.2.2 of MCPM for size of variables
- Old Frame pointer also called *dynamic link*



# Storage of Local and Global Variables

```
int inGlobal;
void chapter12() {
  int inLocal;
  int outLocalA;
  int outLocalB;

  /* initialize */
  inLocal = 5;
  inGlobal = 3;

  /* perform calculations */
  outLocalA = inLocal++ & ~inGlobal;
  outLocalB = (inLocal + inGlobal) - (inLocal -
  inGlobal);
}
```

# Initialization

```
;## # FUNCTION chapter12
;## # FRAME AUTO  (outLocalB) size 2,     offset -6
;## # FRAME AUTO  (outLocalA) size 2,     offset -4
;## # FRAME AUTO  ( inLocal)    size 2,   offset -2
;## # ARG Size(0) Auto Size(6)Context Size(5)
;## # C_SRC :       inLocal = 5;
  mov.w     #0005H,-2[FB]     ;  inLocal
  ._line    19
;## # C_SRC :       inGlobal = 3;
  mov.w     #0003H,_inGlobal
  ._line    22
```

# Assignment

```
;## # C_SRC: outLocalA = inLocal++ & ~inGlobal;
  mov.w     _inGlobal,R0
  not.w     R0
  mov.w     -2[FB],-4[FB]      ;  inLocal  outLocalA
  and.w     R0,-4[FB]          ;  outLocalA
  add.w     #0001H,-2[FB]      ;  inLocal
;## # C_SRC: outLocalB = (inLocal + inGlobal) -
  (inLocal - inGlobal);
  mov.w     -2[FB],R0          ;  inLocal
  add.w     _inGlobal,R0
  mov.w     -2[FB],R1          ;  inLocal
  sub.w     _inGlobal,R1
  sub.w     R1,R0
  mov.w     R0,-6[FB]          ;  outLocalB
```

# Control Structures

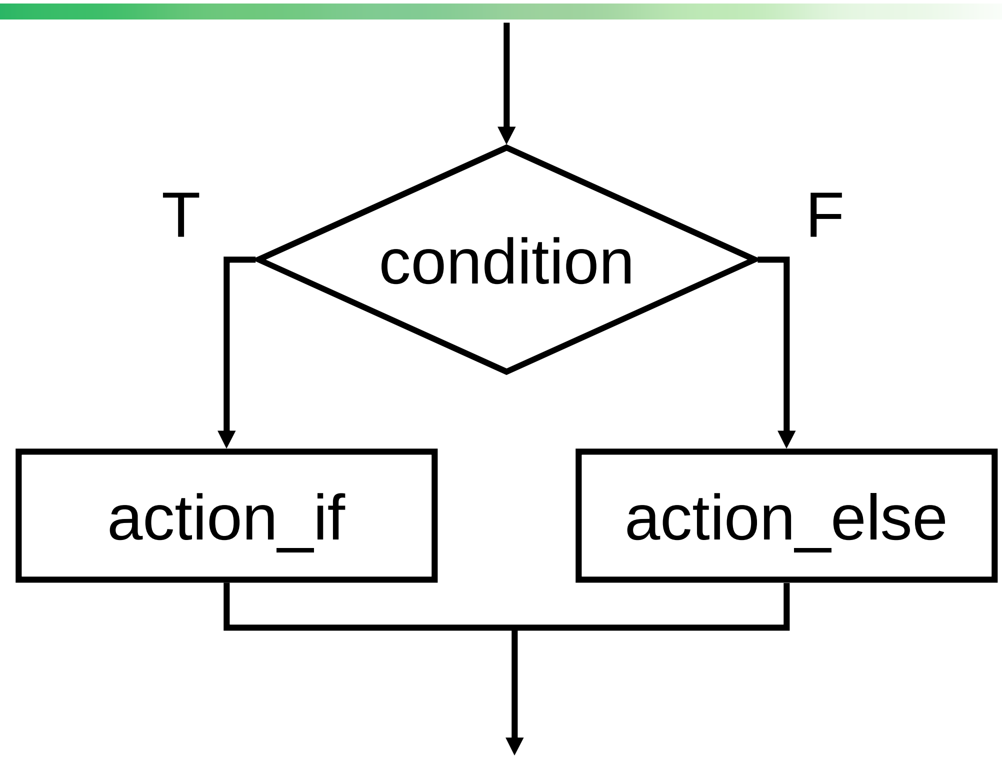
if – else

while loop

for loop

# If-else

```
if (condition)
    action_if;
  else
    action_else;
```




*Else allows choice between*
*two mutually exclusive actions without re-testing condition.*

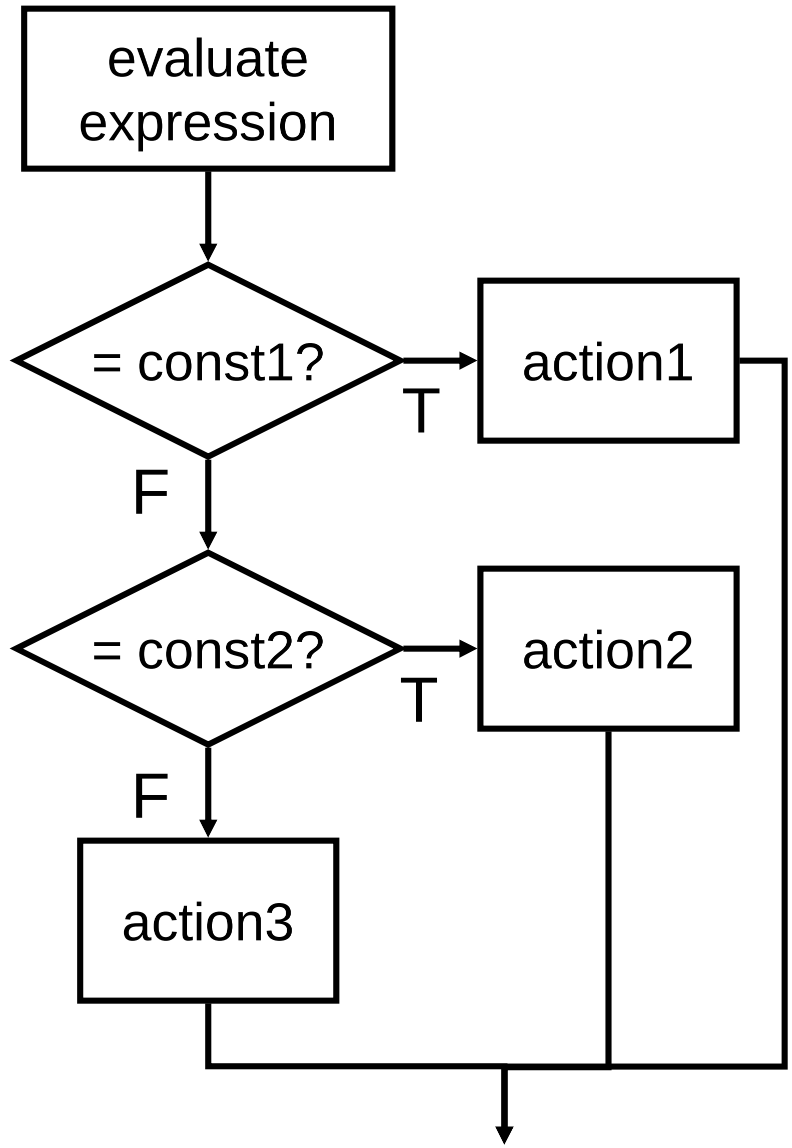
# Generating Code for If-Else

```
if (x){
    y++;
    z--;
  }
else {
    y--;
    z++;
}
```

```
L1:
;## # C_SRC :       if (x) {
        cmp.w #0000H,-6[FB]    ;  x
        jeq   L5
;## # C_SRC :         y++;
        add.w #0001H,-4[FB]    ;  y
;## # C_SRC :         z--;
        sub.w #0001H,-8[FB]    ;  z
;## # C_SRC :       } else {
        jmp   L6
L5:
;## # C_SRC :         y--;
        sub.w #0001H,-4[FB]    ;  y
;## # C_SRC :         z++;
        add.w #0001H,-8[FB]    ;  z
;## # C_SRC :       }
```

# Switch

```
switch (expression) {
  case const1:
    action1; break;
  case const2:
    action2; break;
  default:
    action3;
}
```




*Alternative to long if-else chain.*
*If break is not used, then*
*case "falls through" to the next.*

# Generating Code for Switch

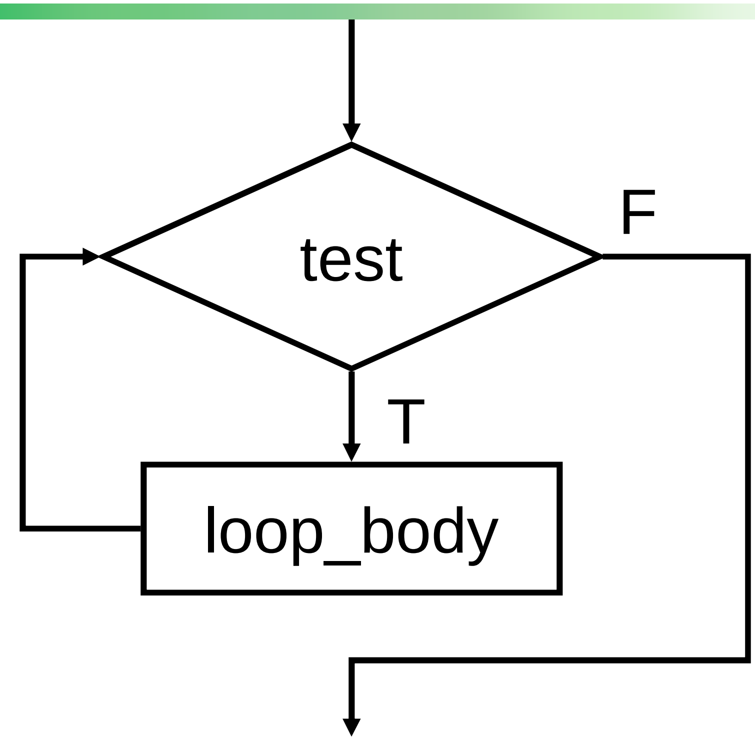
```
switch (x) {
 case 1:
   y += 3;
   break;
 case 31:
   y -= 17;
   break;
 default:
   y--;
   break;
 }
```

```
;## # C_SRC :    switch (x) {
   mov.w        -6[FB],R0 ; x
   cmp.w        #0001H,R0
   jeq  L8
   cmp.w        #001fH,R0
   jeq  L9
   jmp  L10
```

```
;## # C_SRC :    case 1:
L8:
;## # C_SRC :      y += 3;
   add.w        #0003H,-4[FB]  ;  y
;## # C_SRC :      break;
   jmp  L7
;## # C_SRC :    case 31:
L9:
;## # C_SRC :      y -= 17;
   sub.w        #0011H,-4[FB]  ;  y
;## # C_SRC :      break;
   jmp  L7
;## # C_SRC :    default:
L10:
;## # C_SRC :      y--;
   sub.w        #0001H,-4[FB]  ;  y
;## # C_SRC :    }
L7:
;## # C_SRC . . .
```

# While

```
while (test)
    loop_body;
```




*Executes loop body as long as*
*test evaluates to TRUE (non-zero).*

*Note: Test is evaluated **before** executing loop body.*

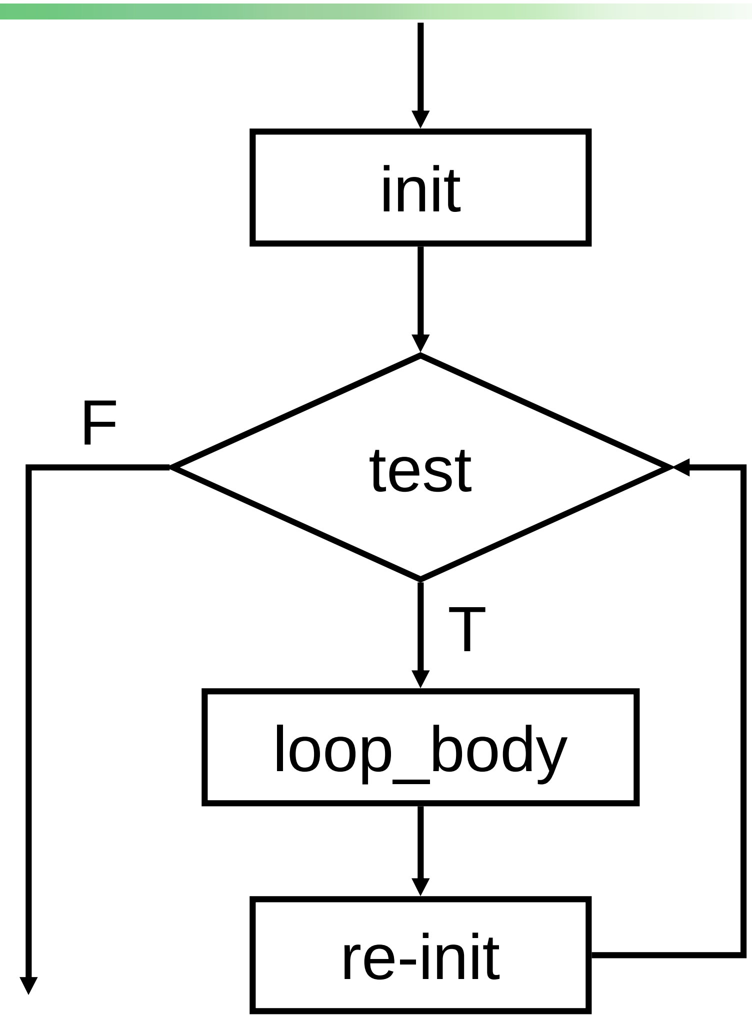
# Generating Code for While

```
x = 0;
 while (x<10) {
   x = x + 1;
 }
```

```
;## # C_SRC :   x = 0;
     mov.w #0000H,-6[FB] ;  x
;## # C_SRC :   while (x < 10) {
L11:
     cmp.w #000aH,-6[FB] ;  x
     jge  L12
;## # C_SRC :     x = x + 1;
     add.w #0001H,-6[FB] ;  x
;## # C_SRC :   }
     jmp   L11
L12: …
```

# For

```
for (init; end-test; re-init)
    statement
```




*Executes loop body as long as test evaluates to TRUE (non-zero). Initialization and re-initialization code included in loop statement.*

*Note: Test is evaluated **before** executing loop body.*

# Generating Code for For

```
for (i = 0; i < 10; i++)
  x += i;
```

```
     ;## # C_SRC : for (i = 0; i < 10; i++) {
          mov.w     #0000H,-8[FB]  ;  i
     L16:
          cmp.w     #000aH,-8[FB]  ;  i
          jge  L18
     ;## # C_SRC :     x += i;
          add.w     -8[FB],-6[FB]  ;  i  x
          add.w     #0001H,-8[FB]  ;  i
          jmp  L16
     L18:
     ;## # C_SRC :  }
```

# ASCII Table

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 00 | nul | 10 | dle | 20 | sp | 30 | 0 | 40 | @ | 50 | P | 60 | ` | 70 | p |
| 01 | soh | 11 | dc1 | 21 | ! | 31 | 1 | 41 | A | 51 | Q | 61 | a | 71 | q |
| 02 | stx | 12 | dc2 | 22 | " | 32 | 2 | 42 | B | 52 | R | 62 | b | 72 | r |
| 03 | etx | 13 | dc3 | 23 | # | 33 | 3 | 43 | C | 53 | S | 63 | c | 73 | s |
| 04 | eot | 14 | dc4 | 24 | $ | 34 | 4 | 44 | D | 54 | T | 64 | d | 74 | t |
| 05 | enq | 15 | nak | 25 | % | 35 | 5 | 45 | E | 55 | U | 65 | e | 75 | u |
| 06 | ack | 16 | syn | 26 | & | 36 | 6 | 46 | F | 56 | V | 66 | f | 76 | v |
| 07 | bel | 17 | etb | 27 | ' | 37 | 7 | 47 | G | 57 | W | 67 | g | 77 | w |
| 08 | bs | 18 | can | 28 | ( | 38 | 8 | 48 | H | 58 | X | 68 | h | 78 | x |
| 09 | ht | 19 | em | 29 | ) | 39 | 9 | 49 | I | 59 | Y | 69 | i | 79 | y |
| 0a | nl | 1a | sub | 2a | * | 3a | : | 4a | J | 5a | Z | 6a | j | 7a | z |
| 0b | vt | 1b | esc | 2b | + | 3b | ; | 4b | K | 5b | [ | 6b | k | 7b | { |
| 0c | np | 1c | fs | 2c | , | 3c | < | 4c | L | 5c | \ | 6c | l | 7c | \| |
| 0d | cr | 1d | gs | 2d | - | 3d | = | 4d | M | 5d | ] | 6d | m | 7d | } |
| 0e | so | 1e | rs | 2e | . | 3e | > | 4e | N | 5e | ^ | 6e | n | 7e | ~ |
| 0f | si | 1f | us | 2f | / | 3f | ? | 4f | O | 5f | _ | 6f | o | 7f | del |

# Masking

One of the most common uses of logical operations is “masking.”

Masking is where you want to examine only a few bits at a time, or modify certain bits.

For example, if I want to know if a certain number is odd or even, I can use an “and” operator.

| | |
|---|---|
| | 0101 0101 0101 0101 |
| AND | <u>0000 0000 0000 0001</u> |
| | 0000 0000 0000 0001 |

Or, lets say you want to look at bits 7 to 2:

| | |
|---|---|
| | 0101 0101 0101 0101 |
| AND | <u>0000 0000 1111 1100</u> |
| | 0000 0000 0101 0100 |

# Example - upper/lower case ASCII

Masking also lets you convert between ASCII upper and lower case letters:

- "A" = 0x41 (0100 0001)
- "a" = 0x61 (0110 0001)

To convert from capitals to lower case:

- Add 32 (0x20)
- OR with 0x20

To convert from lower case to capitals

- Subtract 32 (0x20)
- AND 0xDF

The logical operations are the only way to ensure the conversion will always work