C Programming Language Review and Dissection I:

Overview, Static and Automatic Variables, and Program Control Flow

These lecture notes created by Mr. James B. (Jim) Carlson, NCSU

Today

- High-level review of C concepts

  coupled with . . .

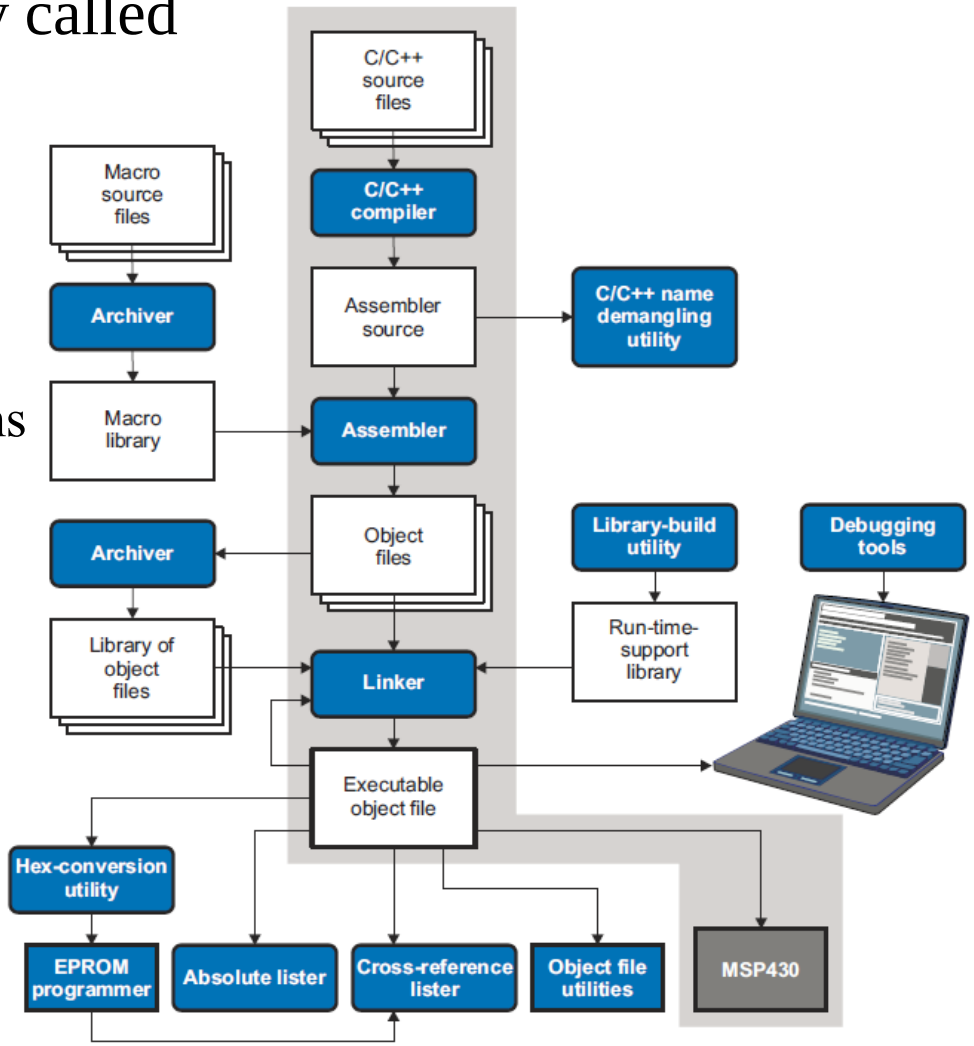
- In-depth examination of *how they are implemented* in assembly language

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

Compiling a C Program

- Entire mechanism is usually called the “compiler”
- Compiler
  - macro substitution
  - conditional compilation
  - “source-level” transformations
    - output is still C
- Assembler
  - generates object file
    - machine instructions
- Linker
  - combine object files (including libraries) into executable image

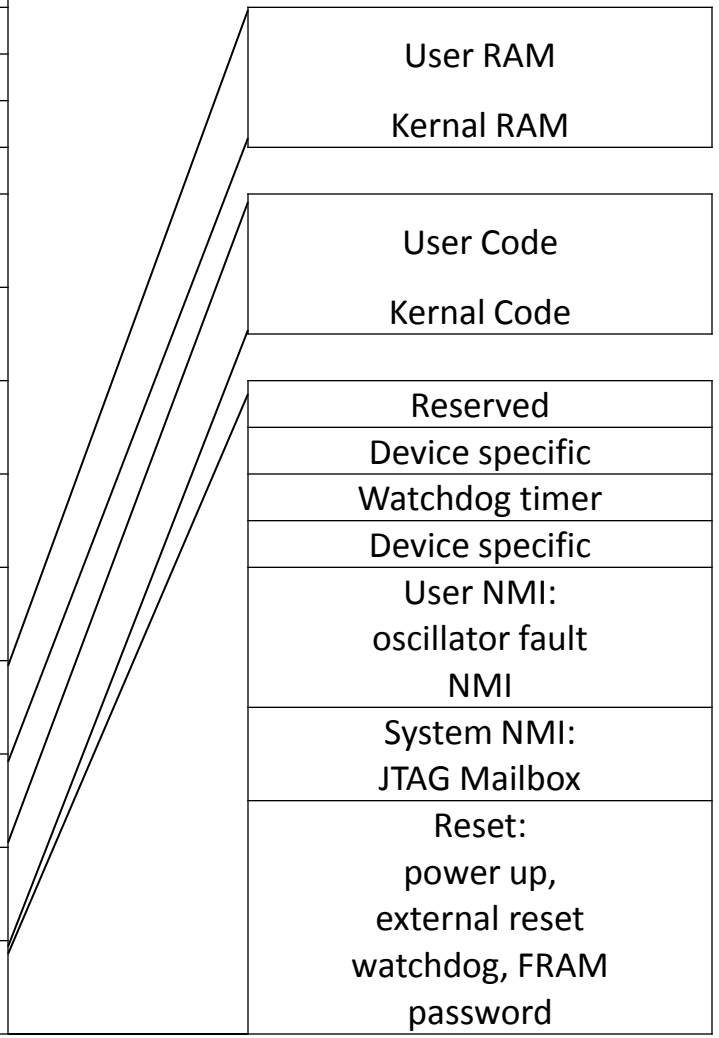


Compiler

- 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

Remember the Memory Map for Our MCU

0x000000	Peripherals
0x000FFF	
0X001000	Boot Strap Loader 0
	Boot Strap Loader 1
	Boot Strap Loader 2
0X0017FF	Boot Strap Loader 3
0X001800	Info B
0X00187F	
0X001880	Info A
0X0018FF	
0X001900	Mirrored to Info B
0X00197F	
0X001980	Mirrored to Info A
0X0019FF	
0X001A00	Device Descriptor Info
0X001A7F	
0X001C00	RAM
0X001FFF	
0X002000	Not Used
0X00C1FF	
0X00C200	Main: Code Memory
0x00FF7F	
0x00FF80	Main: Interrupt Vectors
0x00FFFF	



Classifying Memory Contents

- Variables memory contents (can change as program runs)
  - **Automatic variables**
    - Are declared within a function
    - Only exist between beginning and end of function execution
    - Each time the function is called,
      - Space is allocated for the variable on the stack
      - The function writes the variable's initial value (if any) into that stack location
    - **Located in stack (in the function's stack frame / call stack)**
  - **Static variables [Globals]**
    - Declared outside of all functions, always exist
    - Can make an automatic variable retain its value between invocations by using the "static" keyword
    - **With initial value – located in RAM, but initial value stored in ROM**
    - **Without initial value – located in RAM**
- Fixed memory contents (do not change)
  - Constants and character strings – **located in ROM**
  - Program instructions – **located in ROM**

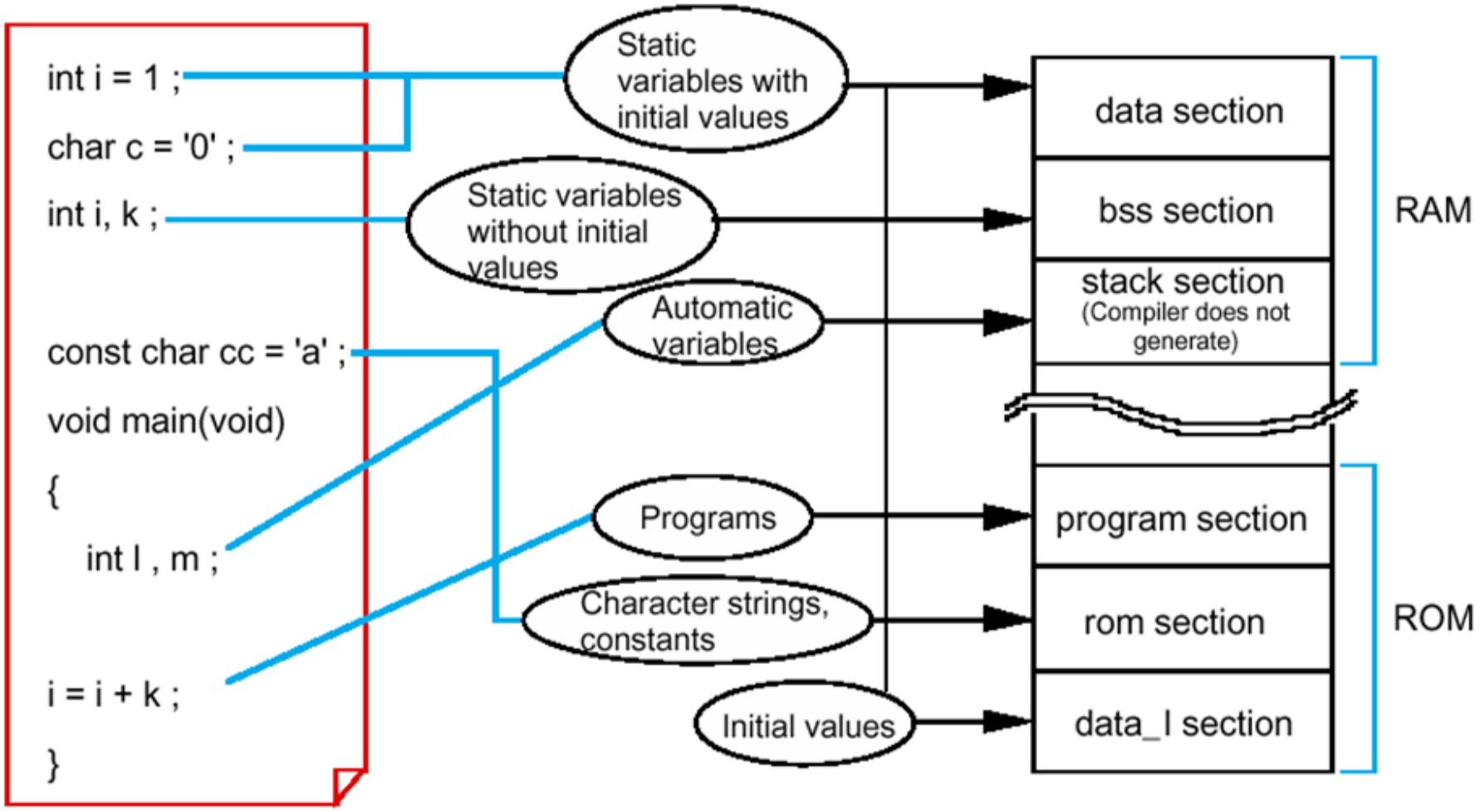
Sections

- The smallest unit of an object file is a *section*.
- A section is a block of code or data that occupies contiguous space in the memory map with other sections.
- Each section of an object file is separate and distinct.
- Object files usually contain three default sections:
  - **.text section contains executable code**
  - **.data section usually contains initialized data**
  - **.bss section usually reserves space for uninitialized variables**

Sections

- In addition, the assembler and linker allow you to create, name, and link *named sections* that are used like
  - .data, .text, and .bss sections.
- There are two basic types of sections:
  - **Initialized sections** contain data or code. The .text and .data sections are initialized; named sections created with the .sect assembler directive are also initialized.
  - **Uninitialized sections** reserve space in the memory map for uninitialized data. The .bss section is uninitialized; named sections created with the .usect assembler directive are also uninitialized.

Sections



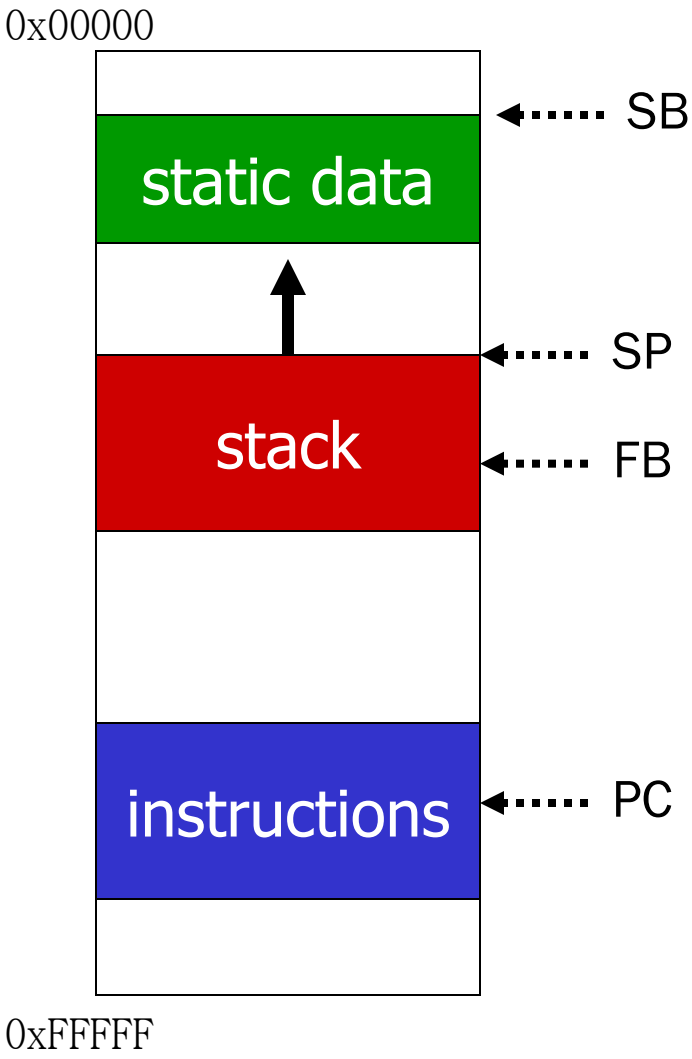
Mapping data into sections by type

Section Names and Contents

- data: static variables which have initial values
- data_I: initial values to load into those static variables
- bss: static variables without initial values ("block started by symbol")
- rom: character strings and constants (located in ROM)
- program: contains program
- fvector: fixed vectors
- vector: variable vectors
- stack: not generated by compiler, but managed at run-time
- heap: not generated by compiler, but managed at run-time

Allocating Space for Variables

- Static data section
  - All static variables stored here (including global variables)
  - There is a fixed (absolute) address. Uses SB (static base) pointer as base to allow shorter (hence faster) instructions.
- 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)



Stack

- The C/C++ compiler uses a function frame stack to:
  - • Allocate local variables
  - • Pass arguments to functions
  - • Save register contents
- The run-time stack grows from the high addresses to the low addresses. The compiler uses the R13 register to manage this stack. R13 is the *stack pointer (SP)*, which points to the next unused location on the stack.
- The linker sets the stack size, creates a global symbol, __STACK_SIZE, and assigns it a value equal to the stack size in bytes. The default stack size is 2048 bytes.
- You can change the stack size at link time by using the

  --stack_size option with the linker command.

Stack

- At system initialization, SP is set to a designated address for the top of the stack. This address if the first location past the end of the .stack section. Since the position of the stack depends on where the .stack section is allocated, the actual address of the stack is determined at link time.
- The C/C++ environment automatically decrements SP at the entry to a function to reserve all the space necessary for the execution of that function. The stack pointer is incremented at the exit of the function to restore the stack to the state before the function was entered.
- If you interface assembly language routines to C/C++ programs, be sure to restore the stack pointer to the same state it was in before the function was entered.

Stack

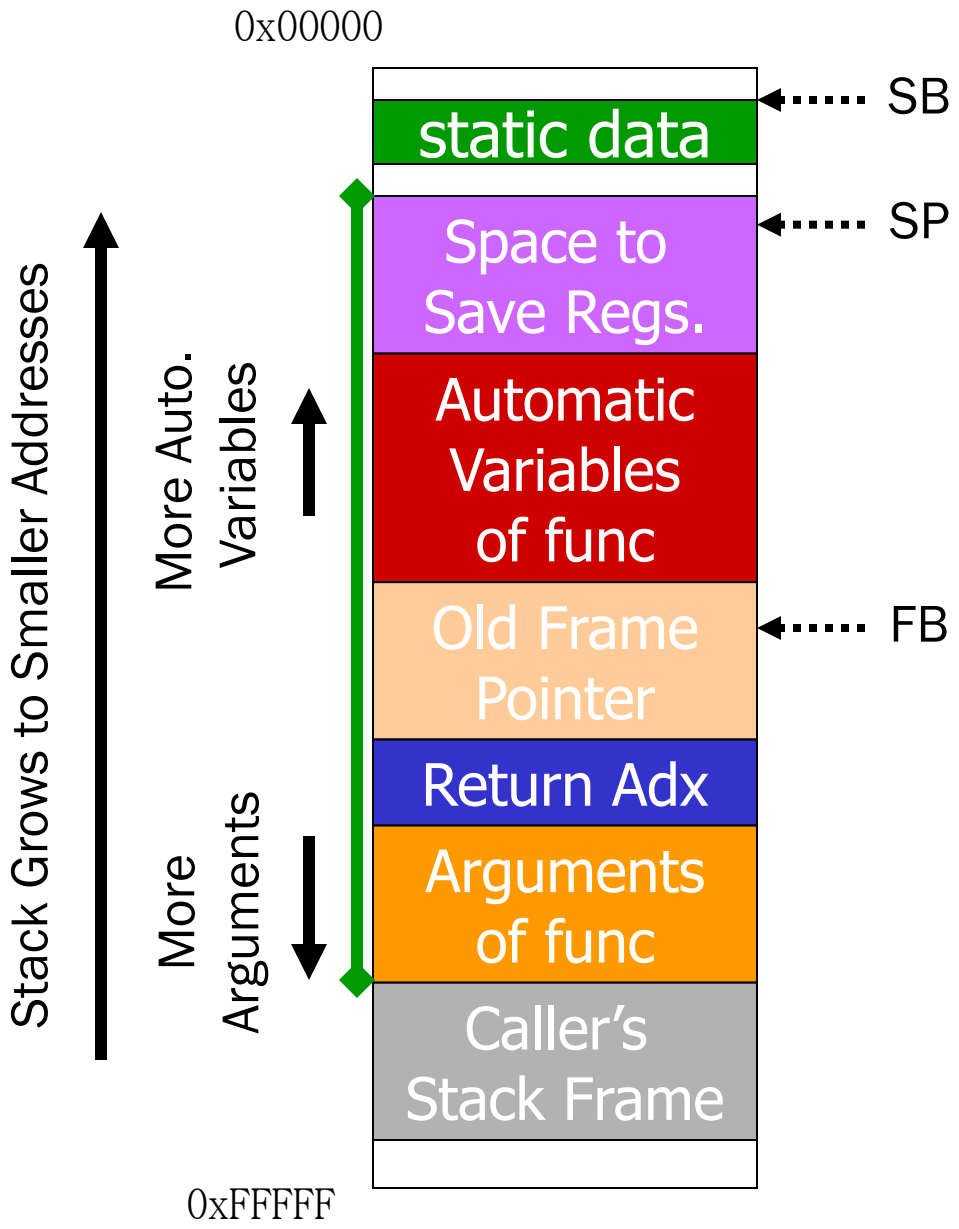
- **Stack Overflow**

- **NOTE: The compiler provides no means to check for stack overflow during compilation or at run time.**

- A stack overflow disrupts the run-time environment, causing your program to fail. Be sure to allow enough space for the stack to grow. You can use the --entry_hook option to add code to the beginning of each function to check for stack overflow;

Activation Record / Function Frame Stack

- Read Patt & Patel Chapter 14 for a thorough explanation of concepts
- Old Frame pointer also called *dynamic link*
- **More on this later**

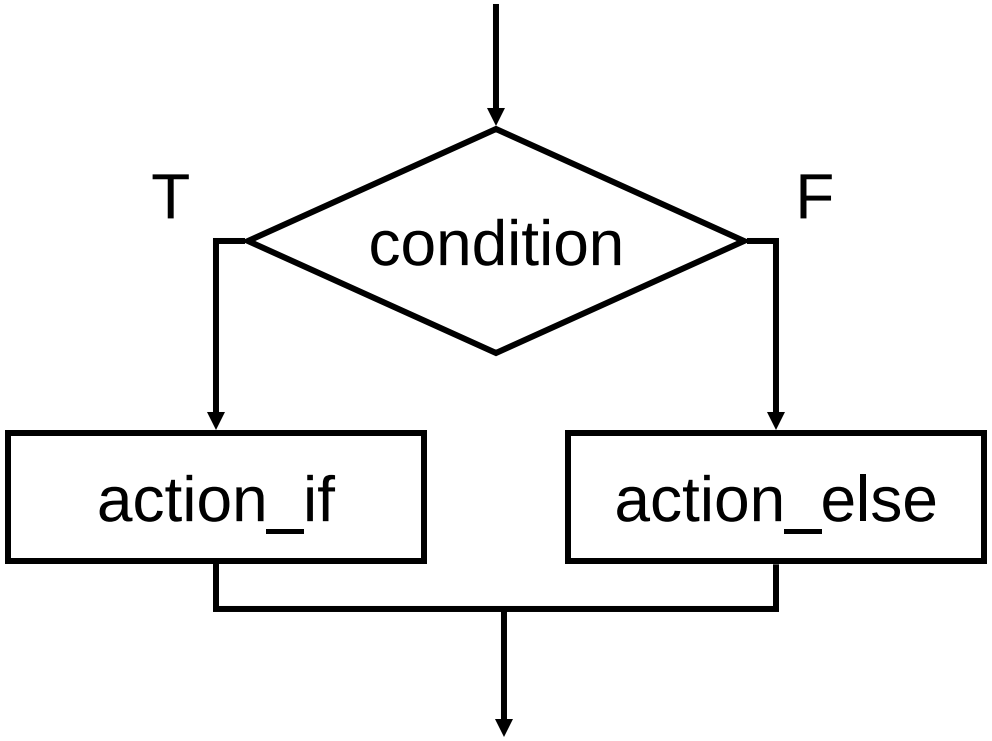


Control Structures

- if – else
- while loop
- for loop

If-else

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

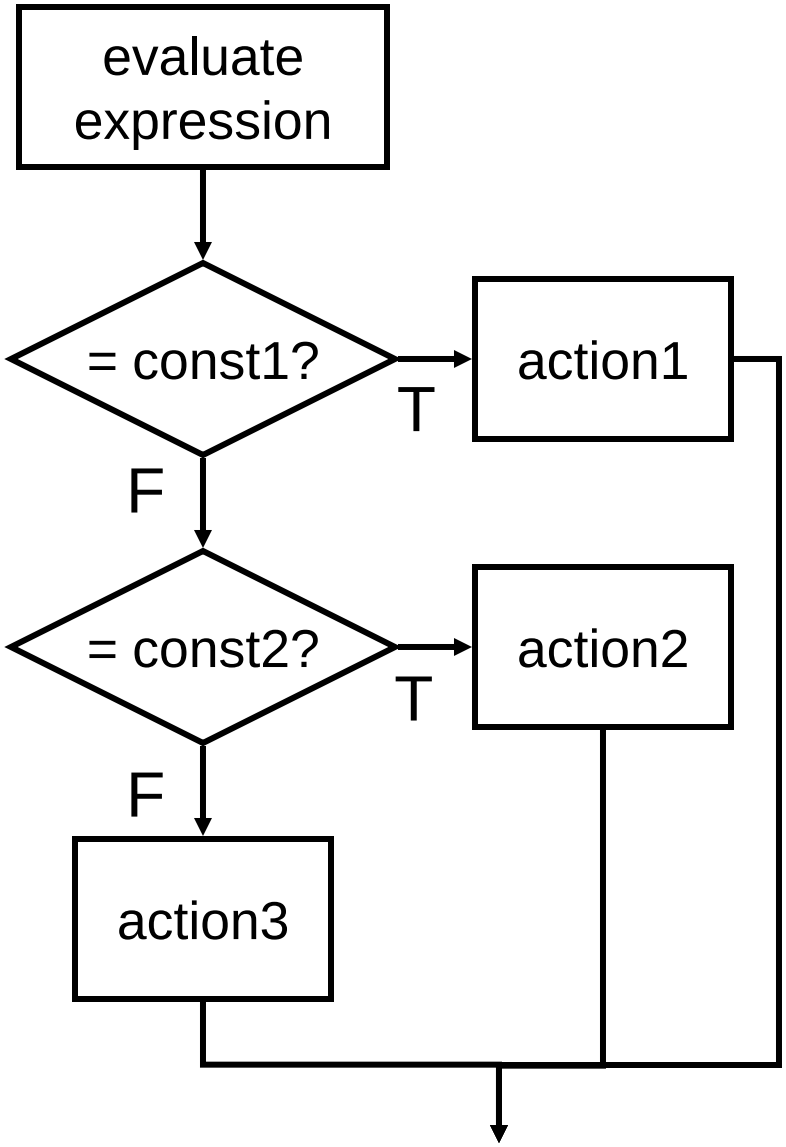


Else allows choice between two mutually exclusive actions without re-testing condition.

Switch

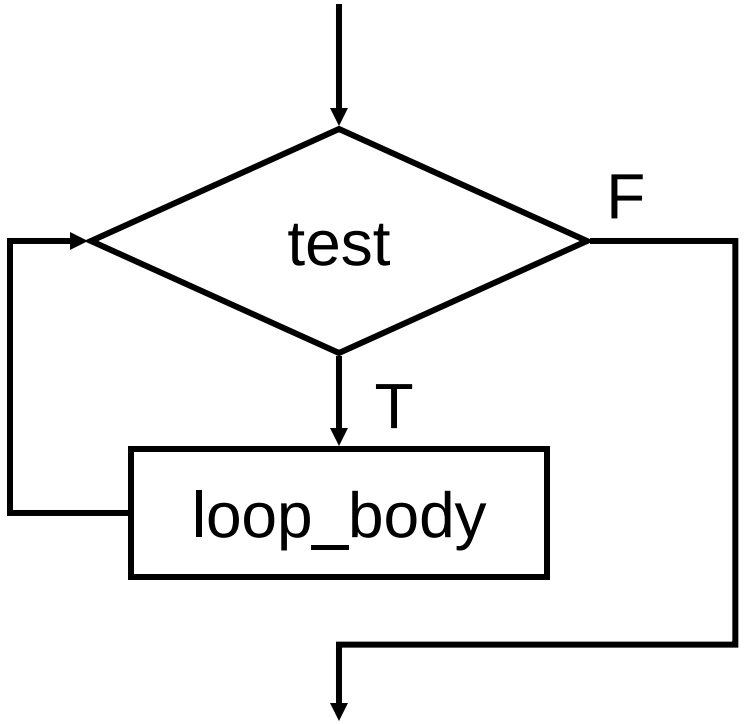
```
• switch (expression) {
  case const1:
    action1; break;
  case const2:
    action2; break;
  default:
    action3;
• }
```

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



While

- **`while (test)`**
  `loop_body;`

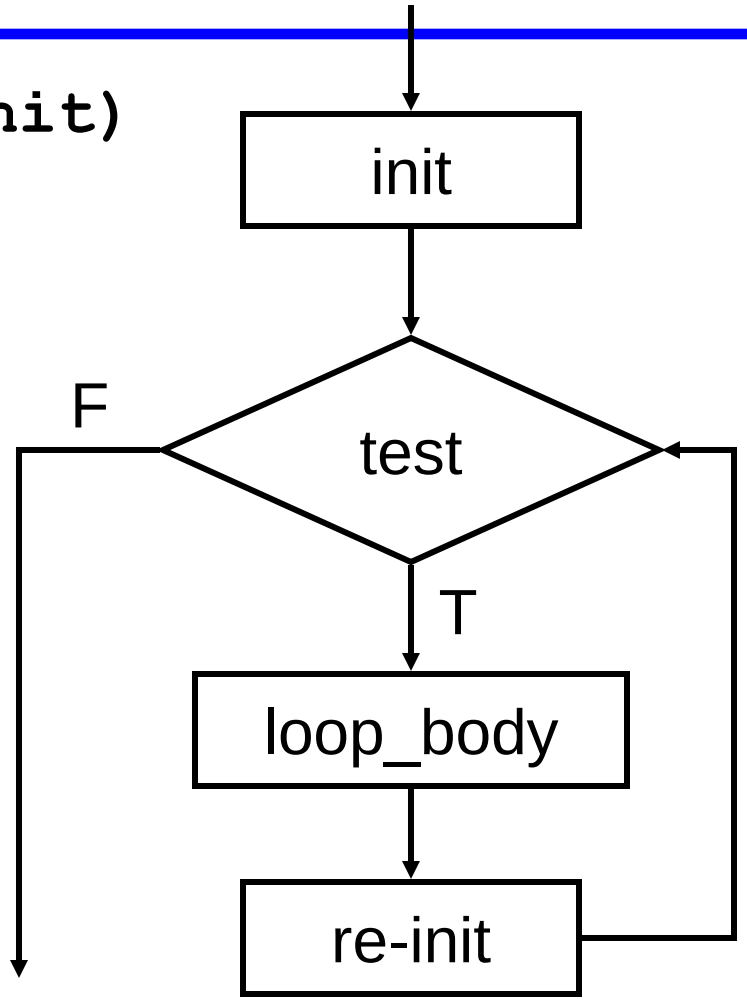


Executes loop body as long as test evaluates to TRUE (non-zero).

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

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.*