Common Concept:

CPU computes address

Memory (scalars and arrays)

address

load (read)

store (write)

address loop variable

index loop variable

lw $t2, 0($t0)

mul $t2, $t0, 4

lw $t2, months($t2)

addi $t0, $t0, 4

addi $t0, $t0, 4

addi $t0, $t0, 1

Seems to require another register for the loop counter
(But you can compare an address to $a0)

Seems to require multiplication and extra instructions.

->> See PH section 3.11 <<-
MIPS Integer Multiply: 2 32-bit factors in, 64 bits output in Hi and Lo registers. Need MIPS mflo to copy Lo to $t2.

Use shift left instead: sll $t2,$t0,2

Shifts and bitwise logic ops are useful for packing and unpacking data.

la  $s1, __start  # $s1=addr of a MIPS instr.
lw  $s1, 0($s1)   # $s1=the MIPS instruction.
andi $s4, $s1, 0xFFFF  #imm is in $s4
srl $s1, $s1, 16     #shift out imm
andi $s3, $s1, 0x1F   #rd is in $s3
srl $s1, $s1, 5      #shift out rd
andi $s2, $s1, 0x1F   #rs is in $s2
srl $s1, $s1, 5      #opcode is in $s1
# to sign extend the immediate value..
sll $s4, $s4, 16
sra $s4, $s4, 16   # $s4=SIGN EXTENDED imm
“System” Stack: “System” initializes $sp and makes some memory at and before $sp usable (stack segment).

You can program with other stacks you create yourself.

    .data
stackLimit: .space 0x10000
stackBottom: .word 0
    .text
# $s7 is our stack pointer
   la     $s7, stackBottom
   ...
   addi   $s7, $s7, -4  #PUSH
   sw     $.., 0($s7)
   ...
   lw     $.., 0($s7)  #POP
   addi   $s7, $s7, 4
#include <iostream.h>
#include <strings.h>
const int MAXWDSIZE = 20;
void reverse(void)
{ static int depth = 0;
  char myword[MAXWDSIZE+1];
  depth++;
  cin >> myword;
  if( strcmp( myword, "." ) != 0 )
    {
      reverse();
      cout << myword << depth << " ";
    }
  depth--;
}
int main(int argc, char *argv[])
{
  reverse();
  cout << "." << endl;
  return 0;
}
Slide 5

```
$ reverse
dog eats cat fast .
fast4 cat3 eats2 dog1 .
$ reverse
dog 123456789012345678901234567890XX cat .
Segmentation fault (core dumped)
$
```

Slide 6

```
Storage Areas: (see PH p.140, Stroustrup p.244)
Stack Frames (activation records):
   LOCAL or AUTOMATIC EXTENT or CLASS
   local variables including procedure parameters
   (could be arrays but usually are scalars)
   linking data
   (return address, bases of adjacent frame)
   NOT for expression evaluation
   unless there are not enough registers.
   (unlike what you are taught
   in data structures courses)

void foo(int param1)
{
    int local1, local2;
    float ftemp[1000]; // possible
    ...
}
Global Data Area: STATIC EXTENT or CLASS
static allocations
  constants (literals–procedure addr’s),
  variables, often arrays.

int BigGlob[1000000];
  // Outside of any function scope
(also function scope vars. declared static)

Free-store or Heap: in C++, NOT intrinsic to C
"PROGRAMMER CONTROLLED", DYNAMIC
dynamic allocations, typically not scalars.

{ ...
  int *pPlus, *p;
pPlus = new int[ 400 ];
p = (int *) calloc(400, sizeof int);
}
  // new makes any constructor be called.
  // calloc zeros the space, malloc and new do not.
(Extents) automatic static
(stack) (data segment)

(Scopes)
Local to function
f() { int V; static int V; }

Local to compilation unit (file)
NONSENSE!! static int V;
(local symbol)

Global (global symbol)
NONSENSE!! extern int V;
//define:
int V;

Note: Good optimizing compilers allocate some automatic variables to REGISTERS when pointers to them are not needed.
free-store
(heap/dynamic)
Storage accessed everywhere through its pointer value.
Pointer variables like all variables can have any scope and extent.