**Data Unlikely to Fit in Registers**

In assembly code,
- programmer identifies data needed and
decides where to put each datum.

Even programs of moderate size are likely to
- have more data
- than registers in the ISA.

In histogram example, we had
- data used to initialize histogram and
data used for counting.
- Could add data for printing, too.

---

**Most Data Not Needed Everywhere in a Program**

A question for you:

**Do all data need to be available in all parts of the program?**

No!

While counting characters for the histogram, we did not need
- initialization data,
- printing data (was not even defined).

---

**A Datum’s Scope Determines Where It Can Be Used**

In assembly code,
- the scope of a datum is
- the part of the program
- in which the datum is accessible.

Usually (in assembly code),
- data are scoped within a subroutine
- or within one “part” of the program (initialization, counting, printing).

Outside of a datum’s scope
- datum does not logically exist
- (bits may still be in location chosen).
C Variables Can Be Global (Whole Program Scope)

But now we're going to write C!

In C,
- we use strings as datum names, so
- we could use a single scope.
- In other words, all data are available in all parts of the code.

Good idea?
Absolutely not!

Picking Lots of Unique Names is Difficult

Why not?
Managing global names is a nightmare.

Imagine a program
- of 1,000,000 lines,
- with 20 programmers.

Can you pick names that are unique?
Did you remember that the program includes library code? What about the names there?

So ... tiny gain, lots of pain.

Use File Scope in C, Not Global Scope

Avoid defining global variables.

C allows programmers to limit scope to
- a file,
- a function, or
- a compound statement.

For file scope,
- put the variable outside of all functions, and
- Write “static” in front of the declaration:

static int my_var; // usable in this file

Local Variables Have Scope within a Function/Block

For function/compound statement scope, declare the variable between braces:

```c
int i, j;
// i and j can be accessed here
```

Such variables are called local variables.
Local Variables Do Not Exist Outside of their Blocks

Usually, local variables are
• created and destroyed
• at the start and end
• of the enclosing function.

{  
  int i, j;  
  // i and j can be accessed here
}

Variables Can be Used without Naming Them

A variable's scope
• defines the part of the program
• in which the variable can be used by
  name.

Using a variable does not require its name.
• You have already seen an example: `scanf`
• If variable exists,
  • its address can be used
  • to read or write the variable.

C Provides Three Storage Classes for Variables

A variable’s storage class defines
• when the variable
  is created and destroyed
• and where in memory
  the variable is stored.

There are three storage classes in C:
• static: exists for the whole program
• automatic: exists for a single block of code
  (such as a function)
• dynamic: created and destroyed on demand

A Memory Map Illustrates Use of Memory

But where are the
storage classes stored?
How do high-level languages (such as C)
make use of LC-3
memory and registers?
Let’s take a look, starting
with a memory map.
Low Addresses are Reserved for the Operating System

Low addresses are usually reserved for the OS.
With LC-3, we have:
- trap vector table at x0000 to x00FF,
- interrupt/exception table at x0100 to x01FF, and
- OS code (trap subroutines).

High Addresses Also Reserved for the Operating System

High addresses are also usually reserved for the OS.
With LC-3, we have memory-mapped I/O at xFE00 to xFFFF.

Code and Data are Mapped After the Low System Area

User code is usually mapped into memory after the system space.
With LC-3, code starts at x3000 by convention.
And program data is mapped into memory after the code.

The Stack is Mapped Above the High System Area

The stack is mapped just above the high system area.
With LC-3, the base can be xFE00, allowing the stack to grow into unused memory.
The Heap is Mapped After the Program’s Data

The heap is mapped just after the global data area. The starting location depends on the size of the program (and data). The heap grows downward into unused memory.

Where are the Storage Classes in Memory?

Static storage class is in global data. R4 points to the top of this region with LC-3. Dynamic storage class is in the heap (program must keep track of variable addresses). Automatic storage class is in the stack.

The Stack Holds One Stack Frame per Function

Let’s look more closely at the stack. R6 points to the top. R5 is a frame pointer. Each function has a stack frame. R5 is a frame pointer.

Stack Frame for main is Pushed First

When a C program starts, the function main is executed (main’s frame pushed on stack).
When `main` Calls `foo`, `foo`'s Stack Frame is Pushed

When a C program starts, the function `main` is executed (`main`'s frame pushed on stack).

`main` may call another function, such as `foo`.

Each Function Call Pushes Another Stack Frame

When a C program starts, the function `main` is executed (`main`'s frame pushed on stack).

`main` may call another function, such as `foo`.

... which calls another ...

R5 always points to the current function's stack frame.
(How, exactly? Later.)

A Function’s Stack Frame is Popped When It Returns

When a function finishes executing, its stack frame is removed from the stack.

Example: One Stack Frame per Function Called

Let’s do an example.

```c
int main ()
{
    int32_t a = 42;
    printf ("%d", a);
    return 0;
}
```

`main`'s stack frame is first.
Example: **main calls printf**

*main calls printf.*

```c
int main ()
{
    int32_t a = 42;
    printf("%d", a);
    return 0;
}
```

Example: **printf calls print_number**

*printf calls print_number.*

```c
int main ()
{
    int32_t a = 42;
    printf("%d", a);
    return 0;
}
```

Example: **print_number calls print_char**

*print_number calls print_char.*

```c
int main ()
{
    int32_t a = 42;
    printf("%d", a);
    return 0;
}
```

Eventually, all functions finish.

Stack and Heap Can Not Be Allowed to Collide

*What happens if the heap and the stack collide?*

As discussed earlier,
- in LC-3/embedded ISA/inside OS, silent data corruption.
- in programs with most ISAs, hardware detects and crashes the program.
What is the LC-3 Calling Convention for R0 through R3?

What about R0 through R3?
Remember: compilers must
◦ systematically generate assembly from C
◦ in a way that matches other compilers’ code.

Are R0, R1, R2, and R3
◦ caller-saved, or
◦ callee-saved?

What is the calling convention?

Assume that R0 through R3 are Caller-Saved

I’m not sure.
I couldn’t find it in the book.
There was once an LC-3 C compiler.
I think R0-R3 were callee-saved.
However, for our class, we will assume:

R0-R3 are caller-saved.

Summary of Static Storage Class

Static variables
◦ part of program’s image on disk
◦ (so can be initialized with bits: 0 by default),
◦ stored in global data area, and
◦ persist for lifetime of program.

Summary of Automatic Storage Class

Automatic variables
◦ created as part of a function’s stack frame,
◦ start as bits
◦ (can optionally by initialized by code), and
◦ destroyed when stack frame is popped
  (end of function execution).
Summary of Dynamic Storage Class

Dynamic variables (How to use? Later.)
- created and destroyed on demand,
- have no names—must be tracked by program, and
- stored in heap.

How C Determines Scope and Storage Class

<table>
<thead>
<tr>
<th>Storage Class</th>
<th>unnamed</th>
<th>function or block scope</th>
<th>file scope</th>
<th>global scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>static (global data)</td>
<td>constants</td>
<td>use 'static' inside block</td>
<td>use 'static' outside of all functions</td>
<td>not 'static' outside of all functions</td>
</tr>
<tr>
<td>automatic (stack)</td>
<td>temporaries, spills</td>
<td>not 'static' inside block</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dynamic (heap)</td>
<td>discussed later</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note that the 'static' qualifier
- changes the scope of variables outside of all functions, but
- changes the storage class of variables inside functions.