ECE 220

Lecture x0007 - 09/17

Slides based on material originally by: Yuting Chen & Thomas Moon

Recap

- Last week:
  - Introduction to C language
  - EWS access & compilation process
  - Variables
    - Identifier, type, scope, storage class, linkage
  - Input and output: `printf`, `scanf`
  - Examples

- Reminders:
  - Midterm 1: on 09/26 at 1900 hrs
    - Practice material posted!
    - Conflict sign-up deadline 09/22
    - HKN review session on 09/22 in ECEB 1002 at 1230 hrs.
  - Quizzes on-going
  - Come to class!

Modified example

- Code snippet on Gitlab: Link

```
#include <stdio.h>

/*Function returns value in celsius
 .... more about functions today */
float to_celsius(int ftemp){
  int celsius; // Local to print_celsius
  celsius = ((ftemp - 32.0)*5.0/9.0);
  return celsius;
}

int main(void){
  int ftemp;  // Local to main
  printf("Enter the temperature in Fahrenheit: ");
  scanf("%d", &ftemp);
  printf("The temperature in Celsius is %.2f\n",
         to_celsius(ftemp));
  return 0;
}
```

What are these # lines called?

What does this one do?

Where does the program start?

Modified example

```c
#include <stdio.h>

/*Function returns value in celsius
 .... more about functions today */

float to_celsius(int ftemp){
  int celsius; // Local to print_celsius
  celsius = ((ftemp - 32.0)*5.0/9.0);
  return celsius;
}

int main(void){
  int ftemp;  // Local to main
  printf("Enter the temperature in Fahrenheit: ");
  scanf("%d", &ftemp);
  printf("The temperature in Celsius is %.2f\n",
         to_celsius(ftemp));
  return 0;
}
```

What is this called?

What is the %d?

What does the & do?

Modified example

```c
#include <stdio.h>

/*Function returns value in celsius
 .... more about functions today */

float to_celsius(int ftemp){
  int celsius; // Local to print_celsius
  celsius = ((ftemp - 32.0)*5.0/9.0);
  return celsius;
}

int main(void){
  int ftemp;  // Local to main
  printf("Enter the temperature in Fahrenheit: ");
  scanf("%d", &ftemp);
  printf("The temperature in Celsius is %.2f\n",
         to_celsius(ftemp));
  return 0;
}
```

`to_celsius` better return a float

`printf` is expecting a float value

What is happening here?

Modified example

```
#include <stdio.h>

/*Function returns value in celsius
 .... more about functions today */
```

```
float to_celsius(int ftemp){
  int celsius; // Local to print_celsius
  celsius = ((ftemp - 32.0)*5.0/9.0);
  return celsius;
}
```

`to_celsius` is a C function and this is its ***function definition.***

```
int main(void){
  int ftemp;  // Local to main
  printf("Enter the temperature in Fahrenheit: ");
  scanf("%d", &ftemp);
  printf("The temperature in Celsius is %.2f\n",
         to_celsius(ftemp));
  return 0;
}
```

Recall subroutines

- ***Functions*** in C are similar to ***subroutines*** in LC3 assembly language

- Both provide abstraction
  - Hides low-level details
  - Gives high-level structure to program, makes it easier to understand overall program flow
  - Enable separable and independent development
  - Reuse code

Math functions vs. C functions

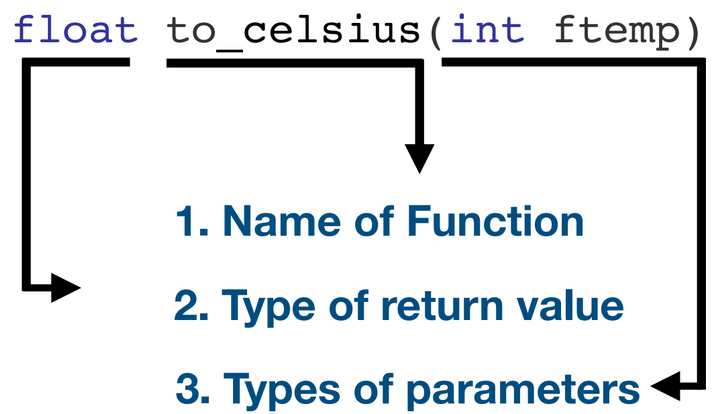
- Recall from Lecture x0002
  - In mathematics, a function $f(x)$ takes a value from a *set* and returns a value in a(nother) *set.* If you call f with some particular value x_0 then it always returns $f(x_0)$.
  - In CS/programming, a function is a piece of code that can be called, *perhaps* with inputs, does some stuff and *maybe* returns something.
  - In *functional* languages (in theory at least), you can replace a function call with its return value and nothing *should* break.

C is **not** a functional language!

C Functions

- Structure of a function
  - Single result returned (optional)
    - Return value is always a particular type
  - Zero or multiple arguments passed in

Function *declaration*



Sometimes also called *function prototype* and *function signature*

Note: Function **declaration** (can be) different from **definition**

Also valid code

Previous slides

Function *declaration/prototype*

```c
#include <stdio.h>

float to_celsius(int ftemp);

int main(void){
  int ftemp;  // Local to main
  printf("Enter the temperature in Fahrenheit: ");
  scanf("%d", &ftemp);
  printf("The temperature in Celsius is %.2f\n",
         to_celsius(ftemp));
  return 0;
}
```

Function *definition*

```c
float to_celsius(int ftemp){
  int celsius; // Local to print_celsius
  celsius = ((ftemp - 32.0)*5.0/9.0);
  return celsius;
}
```

In-fact recommended code

Function *declaration/prototype*

```
#include <stdio.h>

float to_celsius(int ftemp);

int main(void){
  int ftemp;  // Local to main
  printf("Enter the temperature in Fahrenheit: ");
  scanf("%d", &ftemp);
  printf("The temperature in Celsius is %.2f\n",
         to_celsius(ftemp));
  return 0;
}
```

Function *definition*

```
float to_celsius(int ftemp){
  int celsius; // Local to print_celsius
  celsius = ((ftemp - 32.0)*5.0/9.0);
  return celsius;
}
```

Function prototypes

→ Inform the compiler about the properties of functions & allows it to parse `main`

→ Lets a coder delve right into `main` without wading through the function

→ Does not allocate storage for it until it is defined/implemented

→ Allows the compiler to provide optimizations if needed

A quick visualization

https://tinyurl.com/37avn3u5

Exercises

- Write a function which accepts an integer n as an argument and prints out a $n \times n$ identity matrix to the console.
  - Modify the function to make it print out a *lower triangular* **OR** *upper triangular* identity matrix like the ones below (shown for $n = 4$).

$$\begin{bmatrix} 1 & & & \\ 0 & 1 & & \\ 0 & 0 & 1 & \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} 1 & 0 & 0 & 0 \\ & 1 & 0 & 0 \\ & & 1 & 0 \\ & & & 1 \end{bmatrix}$$

Exercises

- We want to do both: upper & lower triangular. What is the logic?
  - Variable `i` iterated over rows, and `j` iterated over columns:

Which part of the matrix is this?

i<j	Upper	Lower
Yes		
No		

- How to specify?
  - Pass in extra argument to specify which type of matrix should be printed, then
  - Use appropriate logic depending on argument

Back to functions: deeper at assembly level

How do functions work at assembly level?

- When C-compiler compiles a program, it keeps track of variables in a program using a **symbol table**.

- For our purposes, the symbol table contains
  - Identifier
  - type of the variable,
  - memory location allocated (by offset - see next slide) and
  - scope

Getting this to work - example

```
int inGlobal=2;
int outGlobal=3;
int dummy(int in1, int in2);

int main(void){
   int x,y,z;
   ...
}

int dummy(int in1, int in2){
   int a,b,c;
   …
}
```

Let us go over this line by line

Getting this to work - example

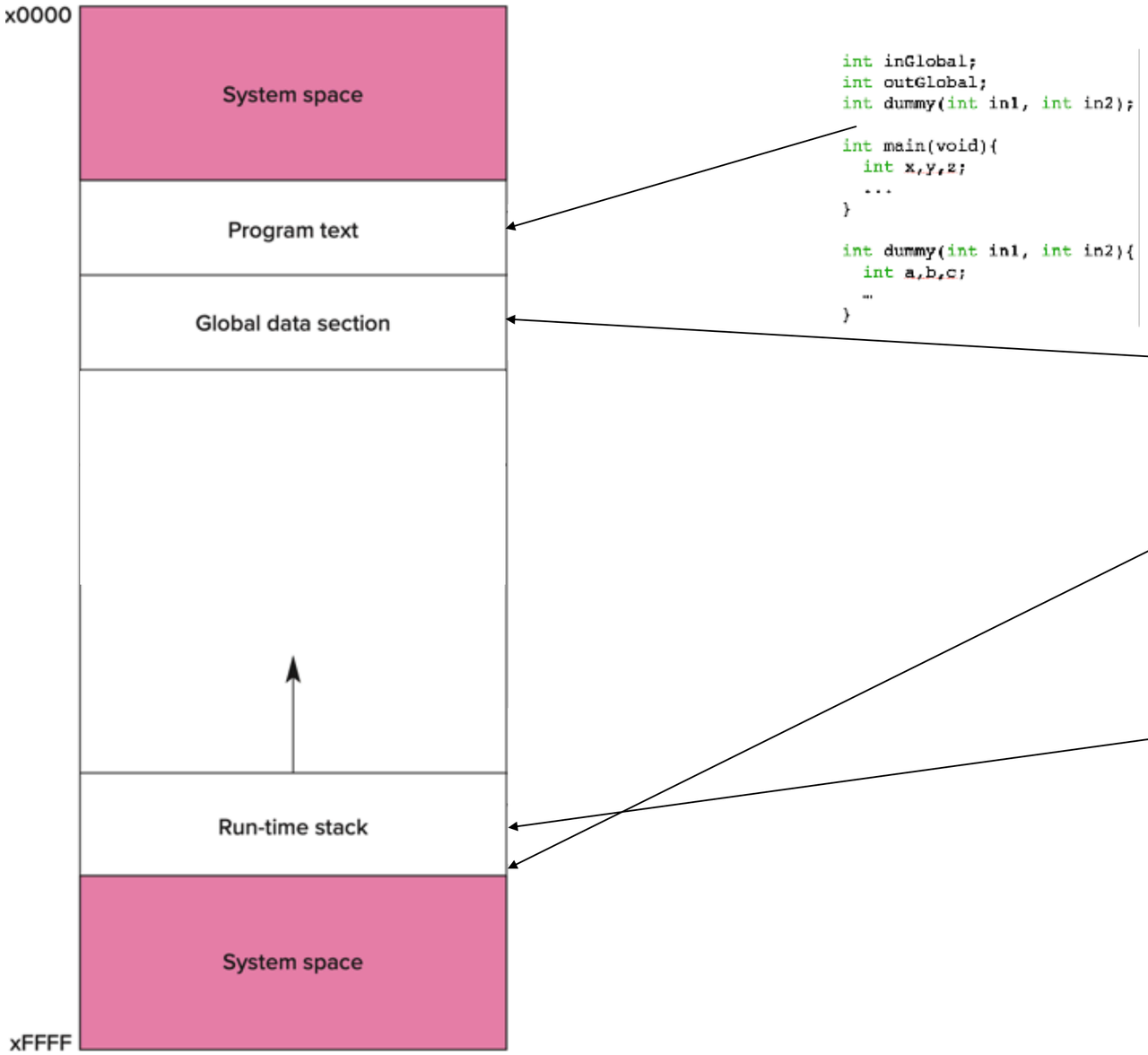
```
int inGlobal=2;
int outGlobal=3;
int dummy(int in1, int in2);

int main(void){
   int x,y,z;
   ...
}

int dummy(int in1, int in2){
   int a,b,c;
   …
}
```

Symbol table

Name	Type	Location	Scope
inGlobal	int	0	Global
outGlobal	int	1	Global
x	int	0	Main
y	int	-1	Main
z	int	-2	Main
a	int	0	Dummy
b	int	-1	Dummy
c	int	-2	Dummy

Where in memory are these stored?

Example: In LC3 memory map




```
int inGlobal;
int outGlobal;
int dummy(int in1, int in2);

int main(void){
  int x,y,z;
  ...
}

int dummy(int in1, int in2){
  int a,b,c;
  …
}
```

Symbol table

Name	Type	Location	Scope
`inGlobal`	int	0	Global
`outGlobal`	int	1	Global
`x`	int	0	Main
`y`	int	-1	Main
`z`	int	-2	Main
`a`	int	0	Dummy
`b`	int	-1	Dummy
`c`	int	-2	Dummy

Some terminology

- **Run-time stack**: A place (actually a stack data structure) to hold *activation frames*
- **Activation frame:** Parts of a *stack* that holds information about <u>each</u> *function call* (sometimes called *stack frames*)*:*
  - Arguments passed in
  - Local variables defined
  - Bookkeeping information

Getting this to work

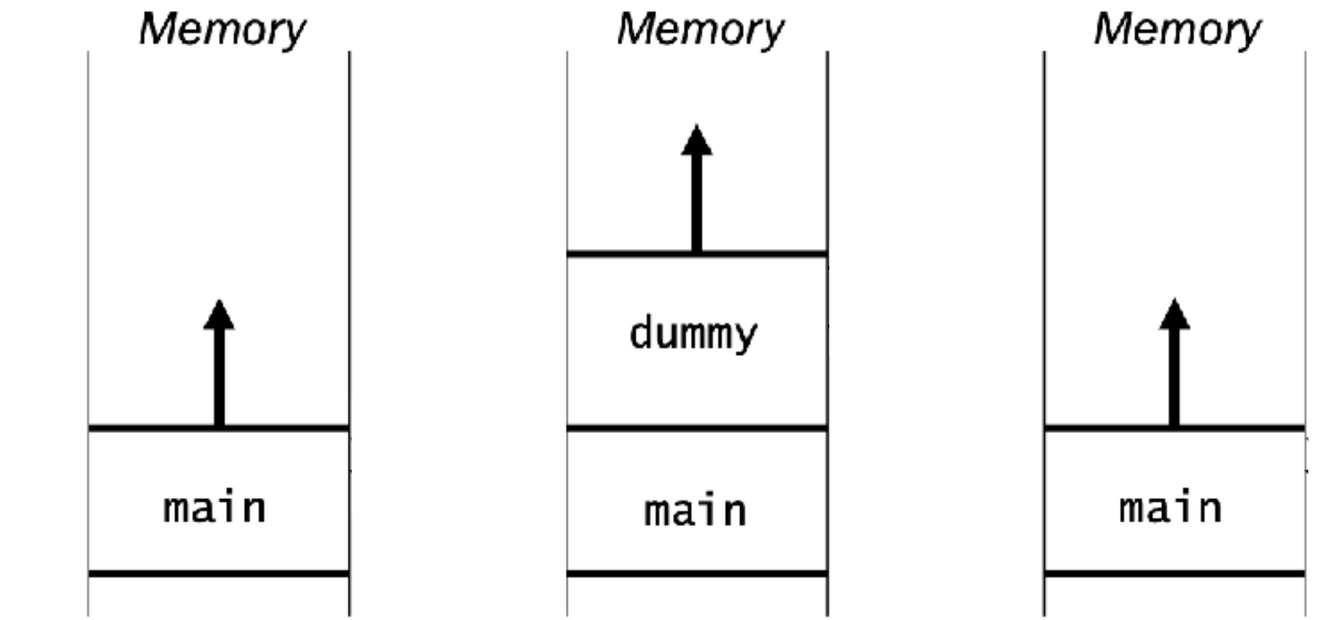
- ***Every*** function *call* creates an activation record (or stack frame) and pushes it onto the run-time stack.

- Whenever a function *completes* (returns), the activation record is popped off the run-time stack

- Whenever a function calls *another one* (nested, including itself), the run time stack grows (pushes another activation record onto the run-time stack).

Example: function call

```
int dummy(int in1, int in2);

int main(void){
   int x,y,z;
   ...
   z = dummy(x, y);
}

int dummy(int in1, int in2){
   int a,b,c;
   …
}
```



Before the dummy call *When dummy executes* *After the dummy completes*

How to keep track?

- Store pointers:
  - Program counter - PC
  - **Global pointer** pointing to first global variable - `R4`
- Top of stack, called **stack pointer** - `R6`
- *Current* **frame pointer** - `R5`
  - Actually points to first local variable of *current* function

Example: global variables

```
int inGlobal=2;
int outGlobal=0;
int dummy(int in1,

int main(void){
  int x,y,z;
  ...
}

int dummy(int in1,
  int a,b,c;
  …
}
```

Name	Type	Location	Scope
inGlobal	int	0	Global
outGlobal	int	1	Global

```
AND R0, R0, #0
ADD R0, R0, #2
STR R0, R4, #0 ; inGlobal=2
AND R0, R0, #0
STR R0, R4, #1 ; outGlobal=0
```

R4 points
the first global variable

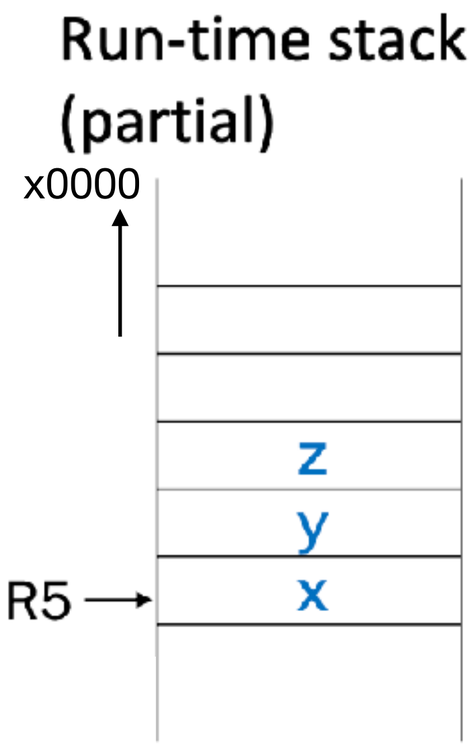
Example: local variables

```
int inGlobal=2;
int outGlobal=0;
int dummy(int in1, int in2);

int main(void){
  int x=3; // Value for e.g.
  int y=0; // Value for e.g.
  ...
}

int dummy(int in1, int in2){
  int a,b,c;
  …
}
```

Name	Type	Location	Scope
inGlobal	int	0	Global
outGlobal	int	1	Global
x	int	0	Main
y	int	-1	Main

```
AND R0, R0, #0
ADD R0, R0, #3
STR R0, R5, #0   ; x = 3
AND R0, R0, #0
STR R0, R5, #-1  ; y = 0
```

Example: local variables



Name	Type	Location	Scope
inGlobal	int	0	Global
outGlobal	int	1	Global
x	int	0	Main
y	int	-1	Main

```
AND R0, R0, #0
ADD R0, R0, #3
STR R0, R5, #0   ; x = 3
AND R0, R0, #0
STR R0, R5, #-1  ; y = 0
```

R5 points
the first local variable

Function calls

- There are four basic steps in the execution of a function call:

    1. argument values from the caller are passed to the callee
    2. control is transferred to the callee
    3. the callee executes its task
    4. control is passed back to the caller, along with a return value

Example: function call

```c
int dummy(int in1, int in2);

int main(void){
  int x,y,z;
  ...
  z = dummy(x, y);
}

int dummy(int in1, int in2){
  int a,b,c;
  …
}
```

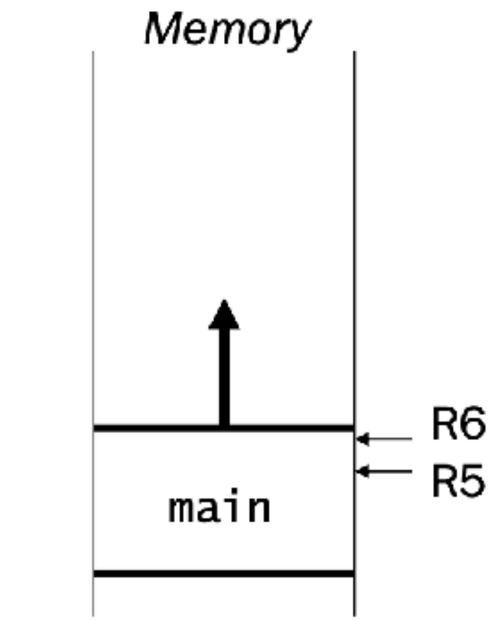
What happens when `main` calls `dummy`?

An *activation record* or *stack-frame* is generated and **pushed** onto the runtime stack & execution transfers to `dummy`

When `dummy` finishes execution its stack-frame is **popped** off and execution transfers back to `main`

Example: function call

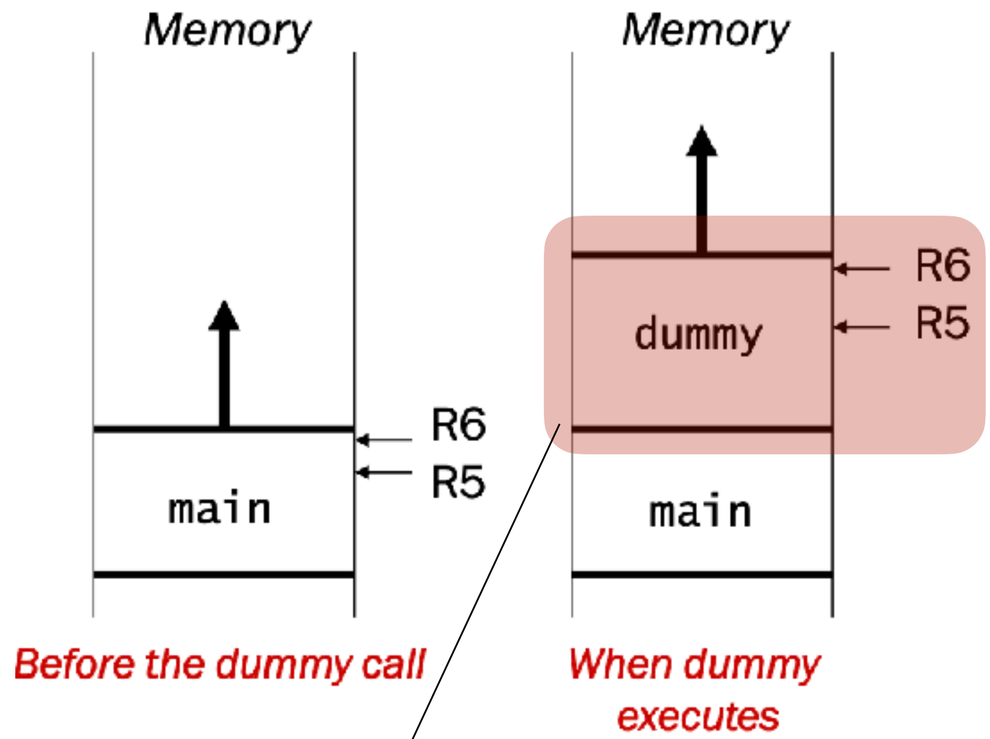
```
int dummy(int in1, int in2);

int main(void){
   int x,y,z;
   ...
   z = dummy(x, y);
}

int dummy(int in1, int in2){
   int a,b,c;
   …
}
```



Before the dummy call

Example: function call

```
int dummy(int in1, int in2);

int main(void){
  int x,y,z;
  ...
  z = dummy(x, y);
}

int dummy(int in1, int in2){
  int a,b,c;
  …
}
```

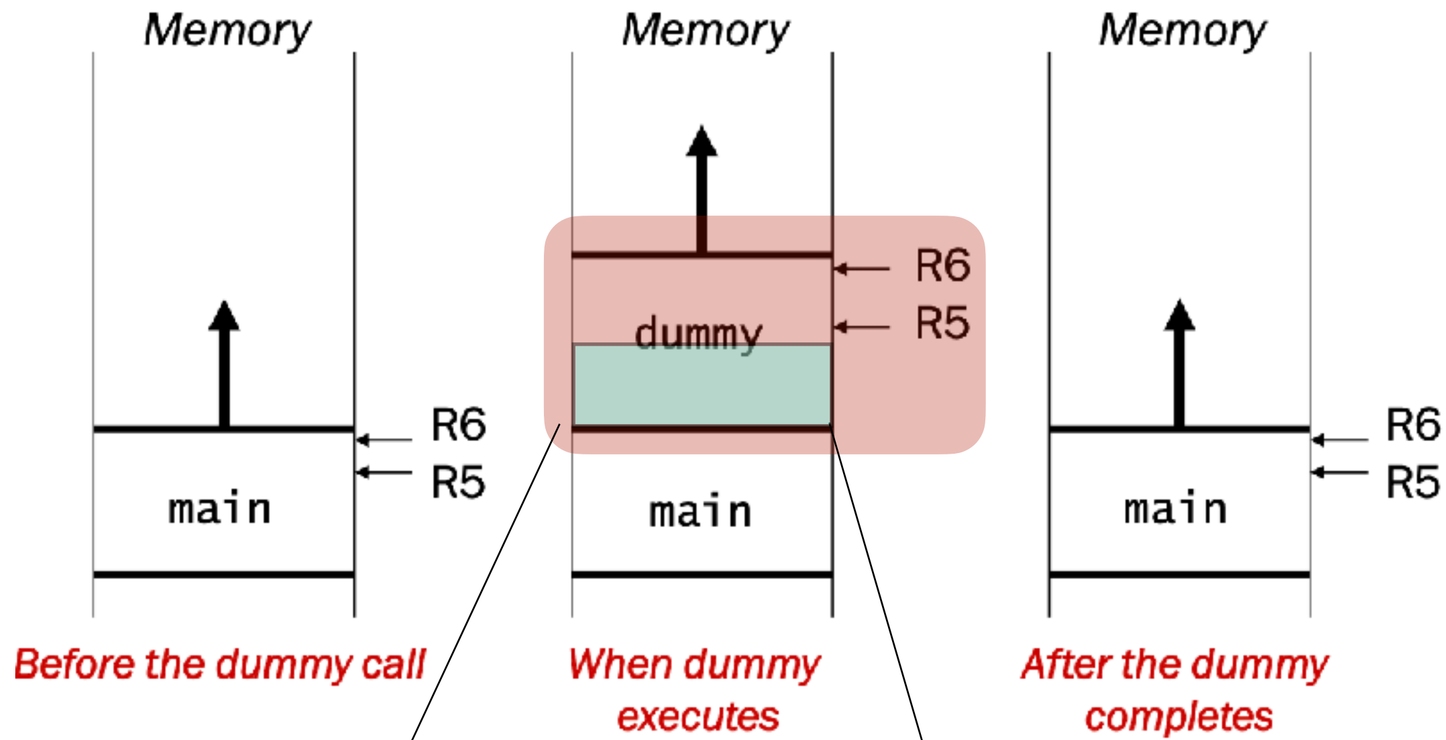


Activation record for dummy

Note: R6 and R5 were duly updated

Example: function call

```
int dummy(int in1, int in2);

int main(void){
  int x,y,z;
  ...
  z = dummy(x, y);
}

int dummy(int in1, int in2){
  int a,b,c;
  …
}
```



Activation record for dummy

If `R5` is first local variable, what goes here?

Details of a function call

- To *successfully* transfer execution between the caller and callee a few things need to be taken care of:
  - Arguments need to be passed around
  - Bookkeeping has to be done:
    - **Return value**: Space for value returned by function according to type has to be allocated
    - **Return address**: Pointer to next instruction has to be saved so caller can resume
    - Caller's frame pointer saved
  - Callee local variables have to be stored

Activation record

Pushed before local variables

Details of a function call

- To *successfully* transfer execution between the caller and callee a few things need to be taken care of:
  - Arguments need to be passed around
  - Bookkeeping has to be done:
    - **Return value**: Space for value returned by function according to type has to be allocated
    - **Return address**: Pointer to next instruction has to be saved so caller can resume
    - Caller's frame pointer saved
  - Callee local variables have to be stored

Activation record

Pushed before local variables

Generating an activation record

- Caller build-up: Push callee's arguments onto stack
- Pass control to callee (`JSR/JSRR`)

Stack build up

Caller

- Callee build-up: (push bookkeeping info and local variables onto stack)
- Execute function
- *Callee tear-down* (update return value, pop local variables, caller's frame pointer and return address from stack)
- Return to caller (`RET`)

Stack teardown

Callee

- *Caller tear-down* (pop callee's return value and arguments from stack)

Caller

Next time

- Stack build-up and stack tear-down

- Examples