ECE 220

Lecture x0007 - 02/10

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:
  - Quizzes on-going
  - Come to class!

Warm-up exercise

- Write a program that prompts and accepts an integer n from the user and then provided that $2 \leq n \leq 8$, prints out an upper or lower triangular portion of the $n \times n$ identity matrix to the console.
- Ask the user which option they want.
- Example $n = 4$ here.

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

Lesson objectives

- Be able to write functions in C
- Understand syntax for function definitions, argument passing and returning in C.
- Understand memory layout conventions & offsets in symbol table
- Understand usage of stacks for function calls and stack build-up and tear-down
- Understand conceptually how to lower simple C functions to LC3

Recall subroutines

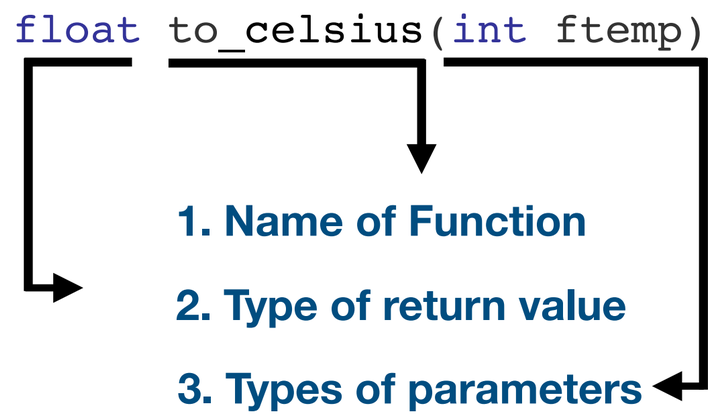
- ***Functions*** in C are similar to ***subroutines*** in LC3 assembly language

- 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

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!**

Warm-up exercise

- Write ***function(s)*** print out an upper or lower triangular portion of the $n \times n$ identity matrix to the console. Then write program that prompts and accepts an integer n from the user and then provided that $2 \leq n \leq 8$, prints out a lower or upper triangular matrix.

- Example $n = 4$ here.

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

How do functions work: 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

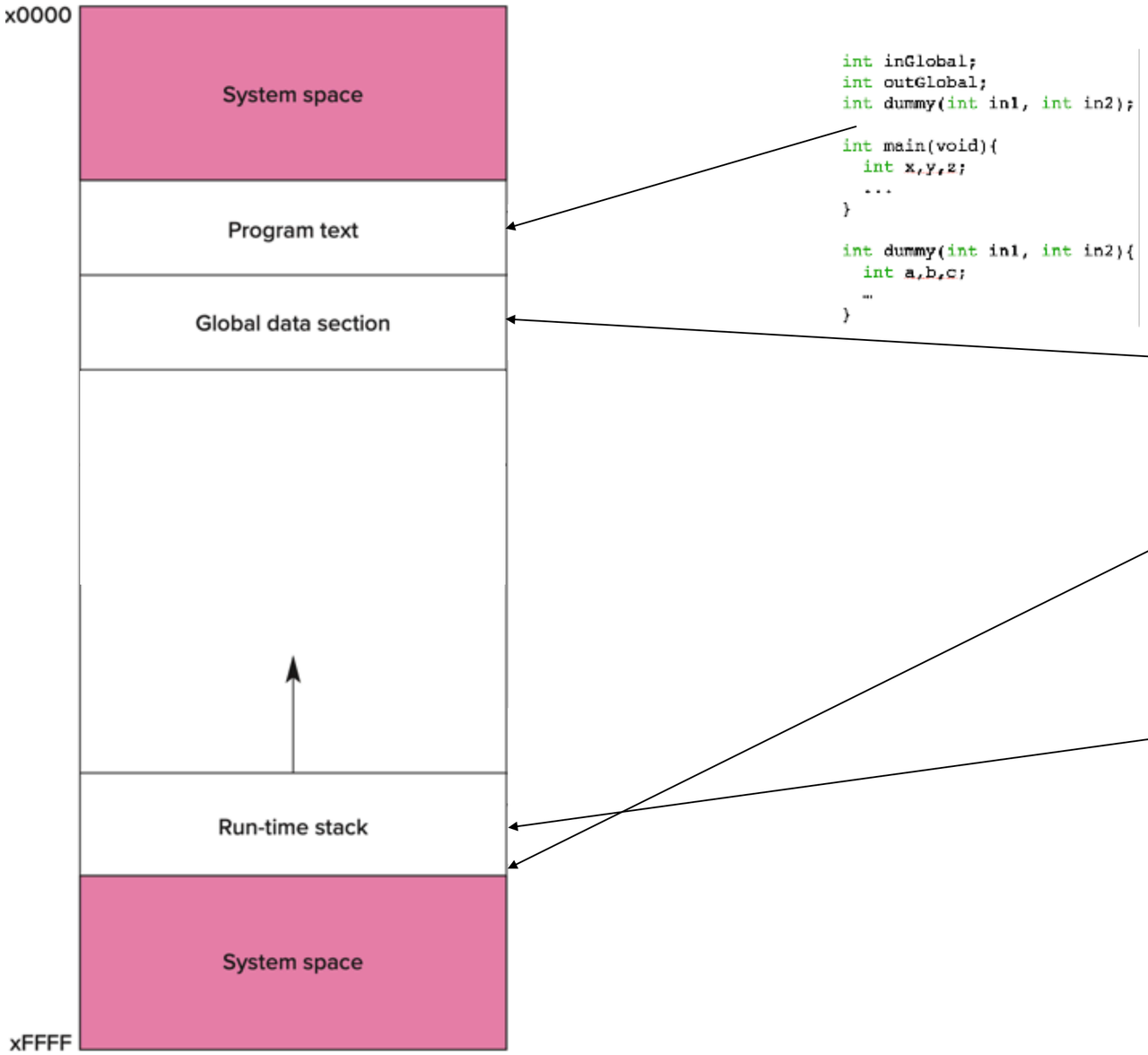
```
int inGlobal=2;
int outGlobal=3;
int dummy(int in1, int in2);

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




```c
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

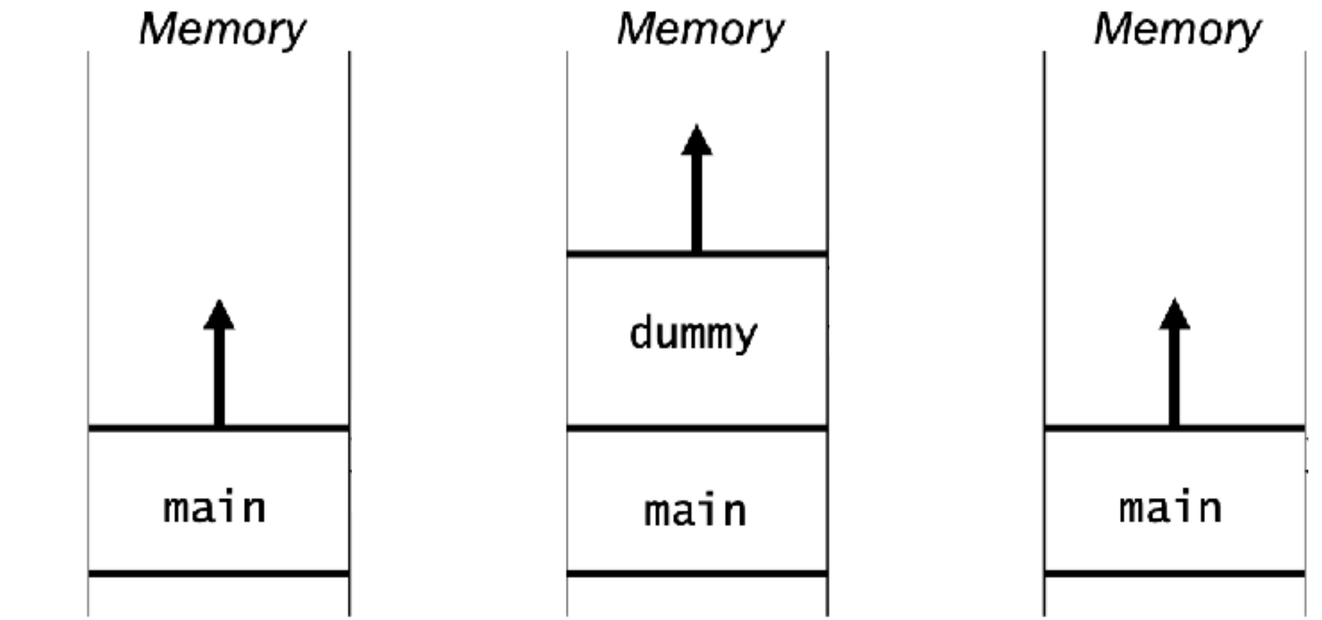
- ***Every*** function *call* creates an activation record (or stack frame) and pushes it onto the run-time stack.

- 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).
- Quick visualization: **https://tinyurl.com/37avn3u5**

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;
   …
}
```



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

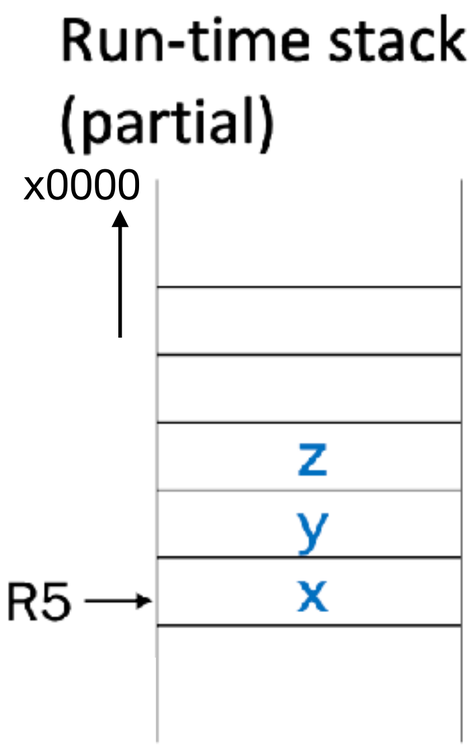
Example: local variables

```
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;
  …
}
```

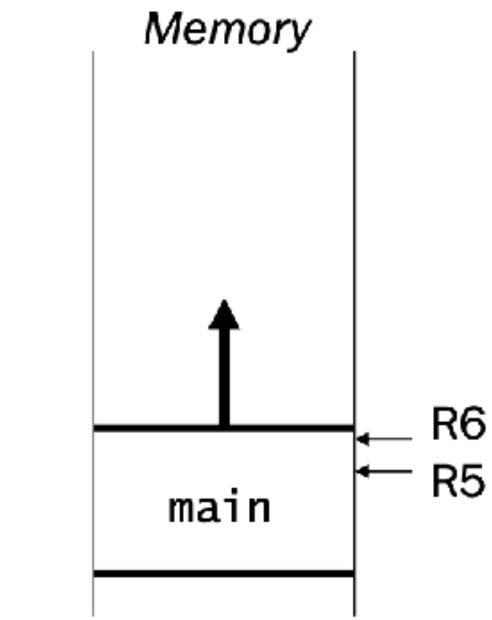
What happens when `main` calls `dummy`?

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

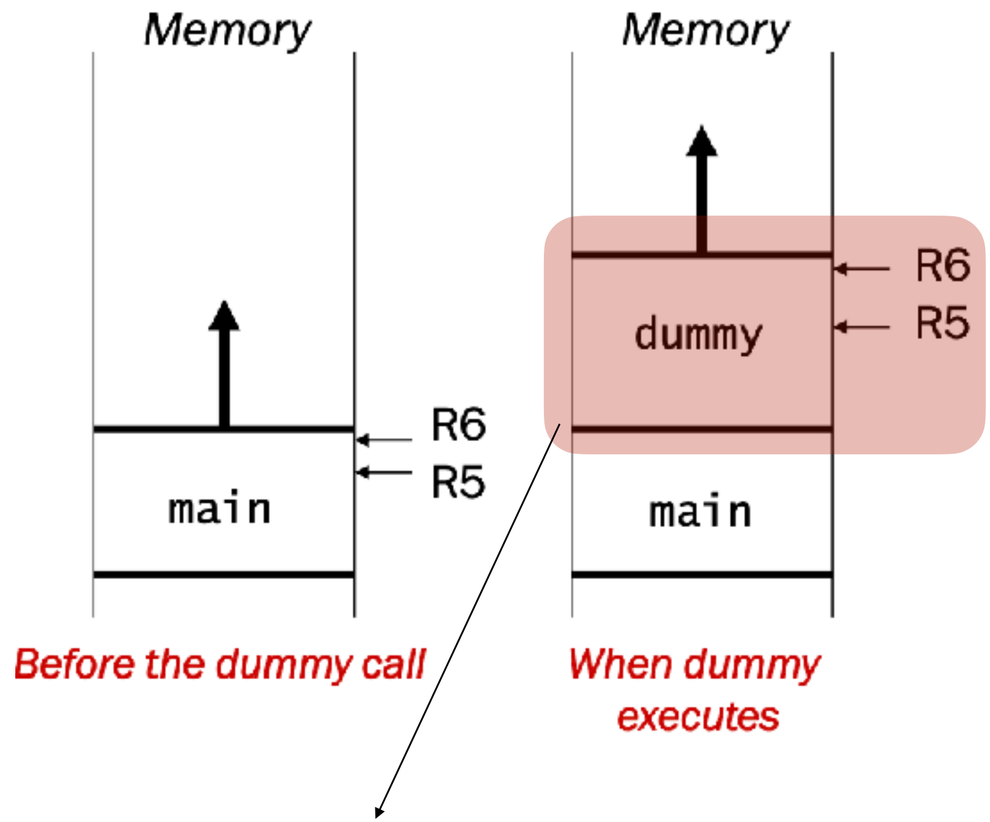
```
int dummy(int in1, int in2);

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

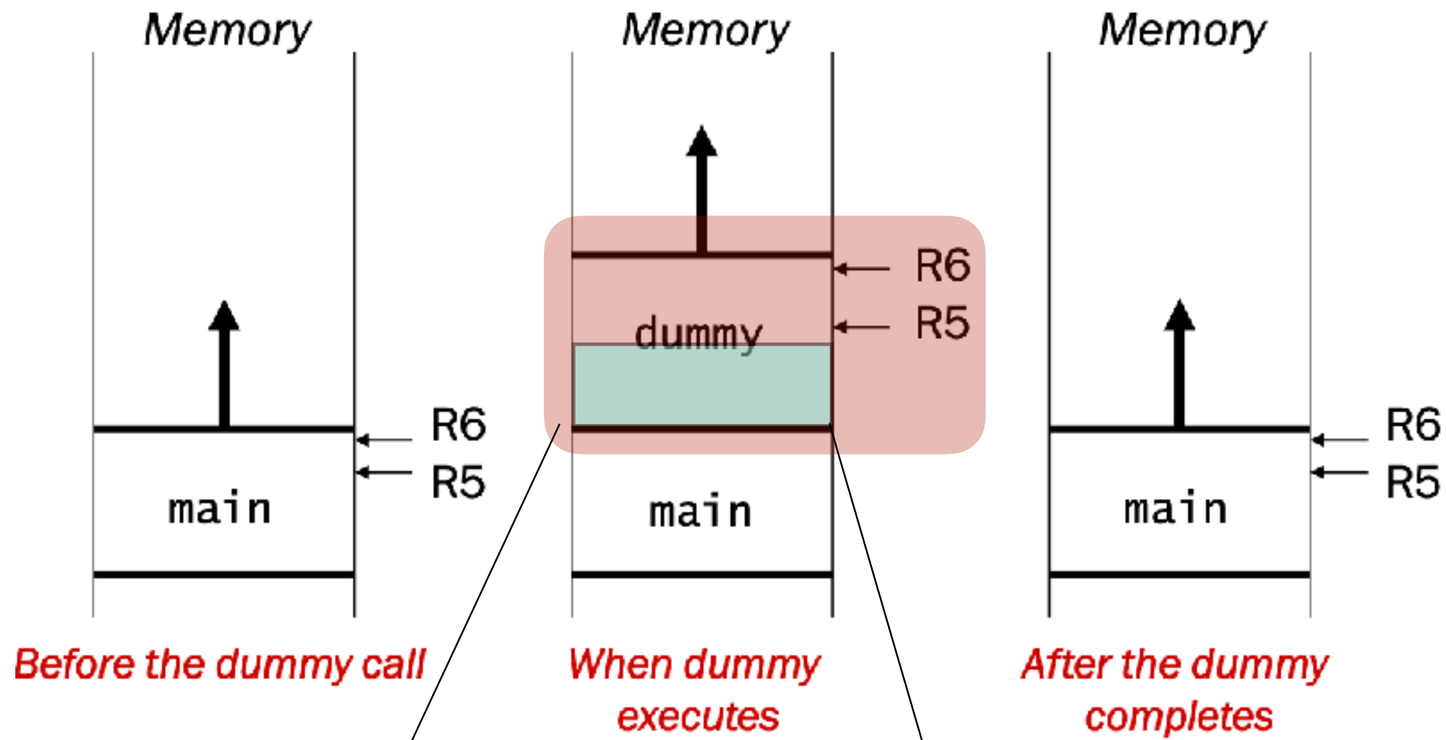
```
int dummy(int in1, int in2);

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

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



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;
  …
}
```



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

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

Stack build up

Callee

- 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

Caller

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

Next time

- Stack build-up and stack tear-down examples for actual C functions.
- Examples.