Functions define named pieces of code

- to whom you can supply values (arguments/parameters)
- which do some computation on those values
- and which return a result

Eg.

```c
int timesTwo(int x) {
    int two_x = x*2;
    return two_x;
}
```
Each function has a signature

- defining the number and types of parameters
- defining the type of the return value

E.g.

```c
// timesTwo takes an int parameter and returns an int result
int timesTwo(int);
```

A function call must supply an appropriate number of values, each with the correct type
Calling Functions

You invoke/call a function

- by giving its name
- by giving values for the parameters
- by using the result

E.g.

```c
int y;
y = timesTwo(2);
```

In fact, C does not require you to use the result of a function
Calling a Function (in more detail)

Example function call

```
res = fun(expr1, expr2, ...)
```

- each expression is evaluated and its value associated to a parameter
- control transfers to the body of the function
- function local variables are created
- the function code executes
- when the result is returned, control returns to the caller
Implementing Functions Calls in MIPS Assembler

When we call a function:

- in the caller code
  - the arguments are evaluated and set up for function (\(a?\))
  - control is transferred to the code for the function (jal fun)

- in code at the start of the function, called the prologue
  - local variables are created (\(t?\))
  - registers to be preserved are saved (\(s?\))

- the code for the function body is then executed

- in code at the end of the function, called the epilogue
  - the return value is set up (\(v0\))
  - control transfers back to where the function was called from (jr $ra)
  - the caller receives the return value
Simple view of implementing function calls in MIPS:

```
main:
    # set params
    # $a0, $a1, ...
    jal func
    # main continues

func:
    ...
    # set return $v0
    jr $ra
```
Function with No Parameters or Return Value

- **jal hello** sets $ra to address of following instruction, and transfers execution to hello
- **jr $ra** transfers execution to the address in $ra

```c
int main(void) {
    hello();
    hello();
    hello();
    return 0;
}

void hello(void) {
    printf("hi\n");
}
```

```asm
main:
    ...
    jal hello
    jal hello
    jal hello
    ...

hello:
    la $a0, string
    li $v0, 4
    syscall
    jr $ra

.data
string:
    .asciiz "hi\n"
```
By convention, function return value is passed back in $v0

```c
int main(void) {
    int a = answer();
    printf("%d\n", a);
    return 0;
}

int answer(void) {
    return 42;
}
```
Function with a Return Value and Parameters

By convention, first 4 parameters are passed in $a0, $a1, $a2, $a3
If there are more parameters they are passed on the stack
Parameters too big to fit in a register, such as structs, also passed on the stack.

```c
int main(void) {
    int a = product(6, 7);
    printf("%d\n", a);
    return 0;
}

int product(int x, int y) {
    return x * y;
}
```
Function calling another function ... DO NOT DO THIS

Functions that do not call other functions - \textit{leaf functions} - are easier to implement.

Function that call other function(s) are harder to implement, because they \textit{must} save $\texttt{ra}$ in their prologue and restore it in their epilogue.

The \texttt{jr $ra} in \texttt{main} below will fail, because \texttt{jal hello} changed $\texttt{ra}$

```c
int main(void) {
    hello();
    return 0;
}

void hello(void) {
    printf("hi\n");
}
```
void f(void);

int main(void) {
    printf("calling function f\n");
    f();
    printf("back from function f\n");
    return 0;
}

void f(void) {
    printf("in function f\n");
}

source code for call_return.c
Simple Function Call Example - broken MIPS

```mips
la $a0, string0  # printf("calling function f
\n");
li $v0, 4
syscall
jal f           # set $ra to following address
la $a0, string1 # printf("back from function f
\n");
li $v0, 4
syscall
li $v0, 0       # fails because $ra changes since main called
jr $ra          # return from function main

f:
la $a0, string2 # printf("in function f
\n");
li $v0, 4
syscall
jr $ra          # return from function f

.data
```

source code for call_return.broken.s

https://www.cse.unsw.edu.au/~cs1521/23T3/
The Stack: Where it is in Memory

Data associated with a function call placed on the stack:

```
0x7FFFFFFFC

Reserved (kernel)

Stack

0x10000000

Data

Program data

0x00400000

Text

Program code

0x00000000

Reserved
```

[https://www.cse.unsw.edu.au/~cs1521/23T3/](https://www.cse.unsw.edu.au/~cs1521/23T3/)
The Stack: Allocating Space

- $sp$ (stack pointer) initialized by operating system
- always 4-byte aligned (divisible by 4)
- points at currently used (4-byte) word
- grows downward (towards smaller addresses)
- a function can do this to allocate 40 bytes:
  
  ```
  sub $sp, $sp, 40  # move stack pointer down
  ```

- a function **must** leave $sp$ at original value
- so if you allocated 40 bytes, before return (**jr** $ra)
  
  ```
  add $sp, $sp, 40  # move stack pointer back
  ```
The Stack: Saving and Restoring Registers - the Hard Way

f:

# function prologue code
sub $sp, $sp, 12  # allocate 12 bytes
sw $ra, 8($sp)   # save $ra on $stack
sw $s1, 4($sp)   # save $s1 on $stack
sw $s0, 0($sp)   # save $s0 on $stack

...  # function body code

# function epilogue code
lw $s0, 0($sp)   # restore $s0 from $stack
lw $s1, 4($sp)   # restore $s1 from $stack
lw $ra, 8($sp)   # restore $ra from $stack
add $sp, $sp, 12 # move stack pointer back
jr $ra  # return
The Stack: Saving and Restoring Registers - the Easy way

f:

# function prologue code
push $ra  # save $ra on $stack
push $s1  # save $s1 on $stack
push $s0  # save $s0 on $stack

...  # function body code

# function epilogue code
pop  $s0  # restore $s0 from $stack
pop  $s1  # restore $s1 from $stack
pop  $ra  # restore $ra from $stack

- note must pop everything push-ed, must be in reverse order
- push & pop are pseudo-instructions
  - push & pop available only on mipsy, not other MIPS emulators
  - but push & pop can be real instructions or pseudo-instructions on other architectures

https://www.cse.unsw.edu.au/~cs1521/23T3/
The Stack: Growing & Shrinking

How stack changes as functions are called and return:
A function that calls another function must save $ra.

```plaintext
main:
    # prologue
    push    $ra    # save $ra on $stack
    jal     hello  # call hello

    # epilogue
    pop     $ra    # recover $ra from $stack
    li      $v0, 0 # return 0
    jr      $ra    #
```
Simple Function Call Example - correct hard way

la  $a0, string0  # printf("calling function f\n");
li  $v0, 4
syscall
jal  f  # set $ra to following address
la  $a0, string1  # printf("back from function f\n");
li  $v0, 4
syscall
lw  $ra, 0($sp)  # recover $ra from $stack
addi $sp, $sp, 4  # move stack pointer back to what it was
li  $v0, 0  # return 0 from function main
jr  $ra  #

f:
la  $a0, string2  # printf("in function f\n");
li  $v0, 4
syscall
jr  $ra  # return from function f

source code for call_return_raw.s
https://www.cse.unsw.edu.au/~cs1521/23T3/
Simple Function Call Example - correct easy way

```assembly
la   $a0, string0    # printf("calling function f\n");
li   $v0, 4
syscall
jal  f               # set $ra to following address
la   $a0, string1    # printf("back from function f\n");
li   $v0, 4
syscall
pop  $ra              # recover $ra from $stack
li   $v0, 0           # return 0 from function main
jr   $ra              #
# f is a leaf function so it doesn't need an epilogue or prologue

f:
la   $a0, string2    # printf("in function f\n");
li   $v0, 4
syscall
jr   $ra             # return from function f
```

source code for call_return.s
https://www.csse.unsw.edu.au/~cs1521/23T3/
MIPS Calling conventions

- \$a0..\$a3 contain first 4 arguments
- \$v0 contains return value
- \$ra contains return address
- if function changes \$sp, \$fp, \$s0..\$s7 it restores their value
- callers assume \$sp, \$fp, \$s0..\$s7 unchanged by call ( jal )
- a function may destroy the value of other registers e.g. \$t0..\$t9
- callers must assume value in e.g. \$t0..\$t9 changed by call ( jal )
MIPS Register usage conventions (not covered in COMP1521)

- floating point registers used to pass/return float/doubles
- similar conventions for saving floating point registers
- stack used to pass arguments after first 4
- stack used to pass arguments which do not fit in register
- stack used to return values which do not fit in register
- for example a struct can be a C function argument or function return value but a struct can be any number of bytes
int answer(void);
int main(void) {
    int a = answer();
    printf("%d\n", a);
    return 0;
}
int answer(void) {
    return 42;
}
# code for function main

main:
    begin
        # move frame pointer
        push $ra
        # save $ra onto stack
        jal answer
        # call answer(), return value will be in $v0
        move $a0, $v0
        # printf("%d", a);
        li $v0, 1
        syscall
        li $a0, '\n'
        # printf("%c", '\n');
        li $v0, 11
        syscall
        pop $ra
        # recover $ra from stack
        end
        # move frame pointer back
        li $v0, 0
        # return
        jr $ra

# code for function answer

answer:
    li $v0, 42
    jr $ra
```c
void two(int i);
int main(void) {
    two(1);
}
void two(int i) {
    if (i < 1000000) {
        two(2 * i);
    }
    printf("%d\n", i);
}
```

source code for two_powerful.c
main:

begin # move frame pointer
push $ra # save $ra onto stack
li $a0, 1
jal two # two(1);
pop $ra # recover $ra from stack
end # move frame pointer back
li $v0, 0 # return 0
jr $ra #

source code for two_powerful.s

https://www.cse.unsw.edu.au/~cs1521/23T3/
Example - Argument & Return - MIPS (two)

two:

    begin       # move frame pointer
    push $ra    # save $ra onto stack
    push $s0    # save $s0 onto stack
    move $s0, $a0
    bge $a0, 1000000, two_end_if
    mul $a0, $a0, 2
    jal two

two_end_if:

    move $a0, $s0
    li $v0, 1    # printf("%d");
    syscall
    li $a0, '\n' # printf("%c", '\n');
    li $v0, 11
    syscall
    pop $s0      # recover $s0 from stack
    pop $ra      # recover $ra from stack
    end          # move frame pointer back
    jr $ra       # return from two

source code for two_powerful.s

https://www.cse.unsw.edu.au/~cs1521/23T3/
```c
int main(void) {
    int z = sum_product(10, 12);
    printf("%d\n", z);
    return 0;
}

int sum_product(int a, int b) {
    int p = product(6, 7);
    return p + a + b;
}

int product(int x, int y) {
    return x * y;
}
```

main:
begin           # move frame pointer
push $ra         # save $ra onto stack
li $a0, 10       # sum_product(10, 12);
li $a1, 12
jal sum_product
move $a0, $v0    # printf("%d", z);
li $v0, 1
syscall
li $a0, '\n'     # printf("%c", '\n');
li $v0, 11
syscall
pop $ra          # recover $ra from stack
end              # move frame pointer back
li $v0, 0        # return 0 from function main
jr $ra           # return from function main
Example - more complex Calls - MIPS (sum_product)

```
sum_product:
    begin                  # move frame pointer
    push $ra               # save $ra onto stack
    push $s0               # save $s0 onto stack
    push $s1               # save $s1 onto stack
    move $s0, $a0          # preserve $a0 for use after function call
    move $s1, $a1          # preserve $a1 for use after function call
    li $a0, 6              # product(6, 7);
    li $a1, 7
    jal product
    add $v0, $v0, $s0      # add a and b to value returned in $v0
    add $v0, $v0, $s1      # and put result in $v0 to be returned
    pop $s1                # recover $s1 from stack
    pop $s0                # recover $s0 from stack
    pop $ra                # recover $ra from stack
    end                    # move frame pointer back
    jr $ra                 # return from sum_product
```

source code for more_calls.s

https://www.cse.unsw.edu.au/~cs1521/23T3/
• a function which doesn’t call other functions is called a **leaf function**
• its code can be simpler...

```c
int product(int x, int y) {
    return x * y;
}
```

source code for more_calls.c

```assembly
mul $v0, $a0, $a1 # return argument * argument 2
jr  $ra #
```

source code for more_calls.s
```c
int main(void) {
    int i = my_strlen("Hello");
    printf("%d\n", i);
    return 0;
}

int my_strlen(char *s) {
    int length = 0;
    while (s[length] != 0) {
        length++;
    }
    return length;
}
```

```c
# Simple C

int main(void) {
    int i = my_strlen("Hello");
    printf("%d\n", i);
    return 0;
}

int my_strlen(char *s) {
    int length = 0;
    loop:;
    if (s[length] == 0) goto end;
        length++;
    goto loop;
end:;
    return length;
}
```

source code for strlen_array.c
source code for strlen_array.simple.c
int main(void) {
    int i = my_strlen("Hello");
    printf("%d\n", i);
    return 0;
}

int my_strlen(char *s) {
    int length = 0;
    while (s[length] != 0) {
        length++;
    }
    return length;
}
Example - strlen using pointer - MIPS (my_strlen)

```assembly
la  $a0, string  # my_strlen("Hello");
jal my_strlen
move $a0, $v0    # printf("%d", i);
li  $v0, 1       
syscall
li  $a0, '\n'    # printf("%c", '\n');
li  $v0, 11      
syscall
pop  $ra         # recover $ra from stack
end              # move frame pointer back
li  $v0, 0       # return 0 from function main
jr  $ra          
```

source code for strlen_arrays

https://www.cse.unsw.edu.au/~cs1521/23T3/
some local (function) variables must be stored on stack

- e.g. variables such as arrays and structs

```c
int main(void) {
    int squares[10];
    int i = 0;
    while (i < 10) {
        squares[i] = i * i;
        i++;
    }
}
```

```assembly
main:
    sub $sp, $sp, 40
    li $t0, 0
loop0:
    mul $t1, $t0, 4
    add $t2, $t1, $sp
    mul $t3, $t0, $t0
    sw $t3, ($t2)
    add $t0, $t0, 1
    b loop0
end0:
```

**source code for squares.s**
int main(void) {
    int i = my_strlen("Hello");
    printf("%d\n", i);
    return 0;
}

int my_strlen(char *s) {
    int length = 0;
    while (s[length] != 0) {
        length++;
    }
    return length;
}
What is a Frame Pointer

- frame pointer $fp$ is a second register pointing to stack
- by convention, set to point at start of stack frame
- provides a fixed point during function code execution
- useful for functions which grow stack (change $sp$) during execution
- makes it easier for debuggers to forensically analyze stack
  - e.g. if you want to print stack backtrace after error
- using a frame pointer is optional - both in COMP1521 and generally
- a frame pointer is often omitted when fast execution or small code a priority
void f(int a) {
    int length;
    scanf("%d", &length);
    int array[length];
    // ... more code ...
    printf("%d\n", a);
}

code source for frame_pointer.c

f:
    # prologue
    sub $sp, $sp, 4
    sw $ra, 0($sp)
    li $v0, 5
    syscall
    # allocate space for
    # array on stack
    mul $t0, $v0, 4
    sub $sp, $sp, $t0
    # ... more code ...
    # epilogue
    # breaks because $sp has changed
    lw $ra, 0($sp)
    add $sp, $sp, 4
    jr $ra

code source for frame_pointer.broken.s

https://www.cse.unsw.edu.au/~cs1521/23T3/
COMP1521 23T3 — MIPS Functions
Example of Frame Pointer Use - Hard Way

```c
void f(int a) {
    int length;
    scanf("%d", &length);
    int array[length];
    // ... more code ...
    printf("%d\n", a);
}
```

Source code for `frame_pointer.c`
Example of Frame Pointer Use - Easy Way

```c
void f(int a) {
    int length;
    scanf("%d", &length);
    int array[length];
    // ... more code ...
    printf("%d\n", a);
}
```

```
f:
    # prologue
    begin
    push $ra
    li $v0, 5
    syscall
    mul $t0, $v0, 4
    sub $sp, $sp, $t0
    # ... more code ...

    # epilogue
    pop $ra
    end
    jr $ra
```

- **begin** & **end** are pseudo-instructions available only on mipsy

[https://www.cse.unsw.edu.au/~cs1521/23T3/](https://www.cse.unsw.edu.au/~cs1521/23T3/)