Introduction to Functions

CS2023 Winter 2004

Outcomes: Introduction to Functions

- "C for Java Programmers", Chapters 7 (beginning)
- After the conclusion of this section you should be able to
  - Write function declarations and definitions in C
  - Understand how the types of parameters and return values are converted when using functions
  - Place function declarations at the beginning of your programs, knowing what purpose they serve
  - Exit from a program from within a function other than `main`

Why Create Functions?

- Reduce complexity
  - Abstraction: hide information so that you won't need to think about it
  - Minimize code size
  - Improve maintainability and correctness
- Avoid duplicate code
- Limiting effect of changes
- Making a section of code readable
  - short functions valuable!

Functions in C

- A C program consists of one or more function definitions, including exactly one that must be called `main`
- The syntax for C functions is the same as the syntax for Java methods
- All functions are *stand-alone*, which means that they are not nested in any other construct, such as a class
- As in Java, parameters are passed *by value*

Function Declaration

A **declaration** merely provides a function prototype: function header (includes the return type and the list of parameters)

```
int power(int x, int n);
```

The declaration does not say anything about the implementation.

The **definition** of a function includes both the function prototype and the function body, that is its implementation.

How Long Should a Function Be

- Muldner (*C for Java Prog.*) recommends one page
- Steve McConnell (*Code Complete*) recommends up to 200 lines long (not including blanks and comments)
- In any case, a *cohesive* function (does one thing well) won't be very long

Review: Passing by value

Advantage: can use parameters as variables in the function without affecting the correponding argument

```
int power(int x, int n)
{
  int i, result = 1;

  for(i = 1; i <= n; i++)
    result *= x;

  return result;
}
```

This can be rewritten as:

Review: Passing by value

```
int power(int x, int n)
{
  int result = 1;

  while (n-- > 0)
    result *= x;

  return result;
}
```

- Disadvantage: difficult to write a function that returns more than one number
  - Need pointers

Return values

- Void functions
  - `void printStuff()`
  - Sometimes called procedures
  - `return` at end not required
- Non-void functions
  - must return a value!
  - using `return` alone will return an undefined value

Void and Conversions

- Definition:

`int f()` is equivalent to `int f(void)`

- Call:

`f();` is equivalent to `(void)f();`

The value of each actual parameter is implicitly converted to the type of the corresponding formal parameter.

The same rules apply to return type conversion.

```
int f(int);
double x = f(1.2);
```

```
/* Function:  maxi
 * Purpose:   find the maximum of its integer
 *  arguments
 * Inputs:    two parameters
 * Returns:   the maximum of parameters
 * Modifies: nothing
 * Error checking: none
 * Sample call: i = maxi(k, 3)
*/
int maxi(int, int);

int maxi(int i, int j)
{
  return i > j ? i : j;
}
```

```c
#define MAX(i,j) ((i) > (j)? (i) : (j))

int maxi(int i, int j);

int main()
{
  float x = 4.5, y = 4.8, max1, max2;

  max1 = MAX(x,y);
  max2 = maxi(x,y);
  printf("Max. of %f and %f is %f?\n", x, y, max1);
  printf("Or is max. of %f and %f %f?\n", x,y,max2);
  return 0;
}

int maxi(int i, int j){
        return i > j ? i : j;
}
```

Default Conversions

- What happens if compiler has not encountered a prototype prior to the call?
- parameters: `float` converted to `double`, `char` and `short` converted to `int`
- Default conversions may not produce desired result!
- Avoid problems by always placing function prototypes before any functions (including `main`) are defined

Default Conversions

What happens?

```
#include <stdio.h>

int main()
{
   int i;

   scanf("%d", &i);
   printf("%g\n", square(i));

   return 0;
}

double square(double x)
{
   return x * x;
}
```

Why bother with function declarations?

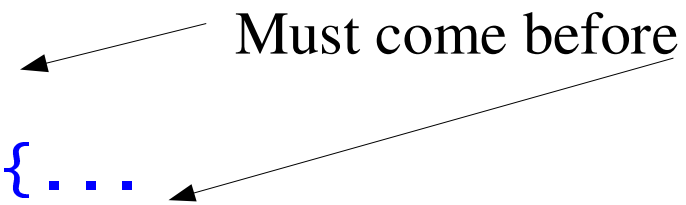
- Could define functions before `main`.
  - Have to be careful of order of function definitions

```
#define N 7
void prn_heading(void) {...}

long power(int m, int n){...}

void prn_tbl_of_powers(int n){...
printf("%ld", power(i, j)); ...}

int main() {
prn_heading();
prn_tbl_of_powers(N);
return 0;
}
```



Function Invocation Means

```
int main(){
float p; int a=5, b=2;
...
p = power(3*a, 2*b);
printf("%f\n", p);
...
}
int power(int x, int n){ ...}
```

1) Each expression in parameter list is evaluated

   `3*a, 2*b`

2) Value of expression is converted, if necessary, to type of formal parameter, and that value is assigned to its corresponding formal parameter at the beginning of the body of the function

   `x = 15, y = 4`

Function Invocation Means (cont'd)

3) The body of the function is executed

```
int i, result = 1;

for(i = 1; i <= n; i++)
  result *= x;

return result;
```

4) If a `return` statement is executed, then control passed back to calling environment

```
p = power(3*a, 2*b);
printf("%f\n", p)
```

Function Invocation Means(cont'd)

5) If the `return` statement includes an expression, then the value of the expression is converted, if necessary, to the type given by the type specifier of the function, and that value is passed back to the calling environment, too.

```
return result; /* Back in main: p = result */
```

6) If the `return` statement does not include an expression, then no useful value is returned to the calling environment

7) If no `return` statement is present, then control is passed back to the calling environment when the end of the body of the function is reached. No useful value is returned.

Function Invocation Means(cont'd)

8) All arguments are passed “call by value”

- Any changes made to formal parameters are lost when control is passed back to calling environment

Exit Function

To terminate the execution of an entire program:

```
exit(int code);
```

Belongs to `<stdlib.h>`, same library that contains `EXIT_SUCCESS` and `EXIT_FAILURE`

`return` equivalent to `exit` in `main`

Exit Function

```
double f(double x) {
 if(x < 0) {
   fprintf(stderr, "negative x\n");
   exit(EXIT_FAILURE); /* no return … */
 }
 return sqrt(x);
}
```