Computer Programming: Skills & Concepts (CP1)
Functions II (Parameters, &, and *)

13th October, 2015
Last lecture, this lecture

Yesterday (Monday 12th Oct)
  ▶ Functions.
  ▶ Examples of simple functions.
  ▶ Program structure and the program environment.

Today (Tuesday 13th Oct)
  ▶ Rules for *declaring* functions.
  ▶ “Scope” of a variable.
  ▶ Parameter-passing in C
  ▶ “Pointers”, &, *.
Declaring functions, revisited

Functions must be *declared* before use.

- But the body of the function does not have to be part of this initial declaration.

You might prefer to read (and write) programs ‘top-down’: high-level structure first, coding details later.

- compiler only needs the function *header* to check it’s correctly used;
- so declare the header first (before *anywhere* the function is called), then define the body of function later (e.g. after the *main* program).

The header is called a function *prototype* or a *type declaration*.

- All the header files like `stdlib.h` and `descartes.h` contain function prototypes, not code.
- The `#include` for these header files is deliberately *right at the top of the program file*.
Example: SumTo again

```c
#include <stdlib.h>
#include <stdio.h>

int SumTo(int n);

int main(void) {
    int n;
    printf("The integer n, please: ");
    scanf("%d", &n);
    printf("sum = %d\n", SumTo(n));
    return EXIT_SUCCESS;
}

int SumTo(int n) {  /* computes 1 + 2+ ... + n */
    int i, sum = 0;
    for (i = 1; i <= n; ++i) {
        sum = sum + i;
    }
    return sum;
}
```

CP Lect 8 – slide 4 – 13th October, 2015
Identifiers are optional in prototypes

```c
#include <stdlib.h>
#include <stdio.h>

int SumTo(int); /* NO NAME NEEDED FOR THE PARAMETER HERE */

int main(void) {
    int n;
    printf("The integer n, please: ");
    scanf("%d", &n);
    printf("sum = %d\n", SumTo(n));
    return EXIT_SUCCESS;
}

int SumTo(int n) { /* NAME DEFINITELY NEEDED HERE */
    int i, sum = 0;
    for (i = 1; i <= n; ++i) {
        sum = sum + i;
    }
    return sum;
}
```

CP Lect 8 – slide 5 – 13th October, 2015
Scope

“Scope” refers to the program sections where a (particular) variable is active/valid:

- global variables are defined above the main function (or indeed any functions) and are valid everywhere.
  - Unless the name of a global variable is re-used for a local one somewhere.
- Local variables are defined within a function and are only valid within that function.
- main is also a function: its variables are only valid there.
- The scope of local variables overshadows the scope of global variables with the same name.
Scope Example

#include <stdio.h>
#include <stdlib.h>
int a = 0;

void f(int n) {
    int i=0; i = i + 1; n = n + 1; a = a + 1;
}

int main(void) {
    int i = 0, n = 0;
    printf("Checkpoint A: i = %d, n = %d and a = %d\n", i, n, a);
    f(n);
    printf("Checkpoint B: i = %d, n = %d and a = %d\n", i, n, a);
    return EXIT_SUCCESS;
}

SCOPE of variables in printfs?

CP Lect 8 – slide 7 – 13th October, 2015
Environment of first Scope Example

---

**Scope I (slide 7)**

- **a)** 1st `printf` [i.e., local to `main`]
  - [a global, so accessible]
  - **Checkpoint A: i = 0, n = 0, a = 0**

- **b)** Call to `foo`
  - [n is local, value 0]
  - **(c) After `foo` is run.**

- **d)** After `foo` finishes, its local environment is discarded

- **e)** 2nd `printf`
  - **Checkpoint B: i = 0, n = 0, a = 1**

---

**Initial Environment**

```
"main"

a: 0
i: 0
n: 0
global var
```
Scope Example 2. (Spot the difference!)

```c
#include <stdio.h>
#include <stdlib.h>
int a = 0, i = 0, n = 0;

void f(int n) {
    int i=0; i = i + 1; n = n + 1; a = a + 1;
}

int main(void) {
    printf("Checkpoint A: i = %d, n = %d and a = %d\n", i, n, a);
    f(n);
    printf("Checkpoint B: i = %d, n = %d and a = %d\n", i, n, a);
    return EXIT_SUCCESS;
}

SCOPE of variables in printfs?

CP Lect 8 – slide 8 – 13th October, 2015
Environment of second Scope Example

Scope II (slide 8)

Initial Environment
- a: 0
- i: 0
- n: 0

Global vars
- main: no local vars

a) 1st printf [i, n, a all global]
   - Checkpoint A: i = 0, n = 0, a = 0

b) Call to fn
   - (call to fn) (only n available to 'main' is global n, copy that)

- fn: 0
- i: 0
- n: 0

"fn initial environment"
- n initialized as the input parameter
- i initialized by local f declaration

"fn call"

"fn initial environment"

After "fn" has finished
- "local beats global"
- So i, n in f-code refer to local variables

c) After fn is run.

d) After f01 finishes it's local environment is discarded

e) 2nd printf
   - Checkpoint B: i = 0, n = 0, a = 1
A closer look at parameters

- We have a function declared (and coded) of type int SumTo(int n)
- We can make calls to this function, eg SumTo(i+2)
- What is the relation between the formal parameter n and the actual parameter i+2?

In C, there is only one way to pass the actual parameter into the real one: call by value - remember discussion in Lecture 7.

This applies to several parameters just as well as to one – each parameter is treated separately.
Call by value

Again, consider `int SumTo(int n)` being called as `SumTo(i+2)`.

- The actual parameter is an expression of a certain type (`int` here).
- The formal parameter is a variable of the same type.

How a function call is evaluated wrt call-by-value:

- The actual parameter is evaluated to yield a value of the specified type. (Whatever the value of `i+2` is.)
- The formal parameter is initialised to that value. (the formal parameter is a local variable of the function body.)
- The function body is executed.
- When `return` is reached, control passes to the point immediately after the function call, and the return value becomes the value of the function call.

Key point: actual parameters are evaluated to values (`int`, `float` etc.) before the function is executed, and the function sees only the values.
Changing variables by function calls

The function only sees the value of parameters. So how can we write a function to swap the values of two variables?

```c
void swap(int a, int b) {
    int temp;
    temp = b;
    b = a;
    a = temp;
}

int main(void) {
    int x = 3, y = 5;
    swap(x,y);
    printf("x is now %d and y is now %d\n",x,y);
    return EXIT_SUCCESS;
}
```

does NOT work!
The magic of & and *

C has a way to use the *address* of a variable (the numbered label of the box for that variable) as a *value*.

If \( x \) is a variable, \&\( x \) is its address.

If \( a \) is an *address*, \*\( a \) means “what is stored at” that address.

And we can store addresses in variables (of type \texttt{int *}).

\begin{verbatim}
int x = 5;  /* x is an int */
int * a;    /* a is the address of an int */
a = &x;     /* Let a have the value of x’s address */
\end{verbatim}

This defines and assigns an int called \( x \), defines a “pointer” called \( a \), and assigns the *value of* \( a \) to be the address of \( x \).
The magic of & and *

```c
int x = 5;  /* x is an int */
int * a;    /* a is the address of an int */
a = &x;     /* Let a have the value of x’s address */
```

After executing the above lines, then assigning to *a is the same as assigning to x and evaluating *a is the same as evaluating x.

A sane language would say type &int instead of type int *.
C programmers usually write int *a; rather than int * a;

Variables of type int * are called pointers to integers. Other pointer variables might be float * etc.
The magic of & and * - Environment

Specific addresses will be different for different runs.

```
int x = 5;  /**< int variable */
int * a;  /**< pointer to int */
```

```
int x = 5;  /**< int variable */
int * a;  /**< pointer to int */
```

```
| a = &x;      | /* assign to a, the specific address of variable x */ |
```

```
int x = 5;  /**< int variable */
int * a;  /**< pointer to int */
```

```
printf("a has value %p, points to value %d. In", a, *a);
```

```
| a = &x;      | /* assign to a, the specific address of variable x */ |
```

```
int x = 5;  /**< int variable */
int * a;  /**< pointer to int */
```

```
printf("a has value %p, points to value %d. In", a, *a);
```

```
| a = &x;      | /* assign to a, the specific address of variable x */ |
```

```
int x = 5;  /**< int variable */
int * a;  /**< pointer to int */
```

```
printf("a has value %p, points to value %d. In", a, *a);
```

```
| a = &x;      | /* assign to a, the specific address of variable x */ |
```

```
int x = 5;  /**< int variable */
int * a;  /**< pointer to int */
```

```
printf("a has value %p, points to value %d. In", a, *a);
```

```
| a = &x;      | /* assign to a, the specific address of variable x */ |
```

```
int x = 5;  /**< int variable */
int * a;  /**< pointer to int */
```

```
printf("a has value %p, points to value %d. In", a, *a);
```

```
| a = &x;      | /* assign to a, the specific address of variable x */ |
```
Swapping variables with & and *

```c
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main(void) {
    int i = 1, j = 2;
    printf("Checkpoint A: i = %d and j = %d.\n", i, j);
    swap(&i, &j);
    printf("Checkpoint B: i = %d and j = %d.\n", i, j);
    return EXIT_SUCCESS;
}
```

Using the combination of & and * we achieve the effect of call by reference – allowing the function to get at the variable itself, not just its value.

*CP Lect 8 – slide 14 – 13th October, 2015*
Swapping with & and * - Environment

Swapping Variables [for slide 14]

(a) Initial Environment

```
"main"
0x7fff44edbo1c 1 "i"
0x7fff44edbo18 2 "j"
```

take value of "address of i" and that is input parameter

(b) After calling `swap(&i, &j)`, i.e.

```
swap(0x7fff44edbo1c, 0x7fff44edbo18)
```

Environment after call

```
"main"
0x7fff44edbo1c 1 "i"
0x7fff44edbo18 2 "j"
```

Alternative view

```
"main"
"i" 1
"j" 2
```

```
"swap"
"a"
"b"
"temp"
```

a and b are (local) input parameters, their value (addresses) gets copied based on call to swap.

temp is initially uninitialized.
Overview: Uses of & and *

```c
int *p;
Definition of a pointer variable
p = &a;
Take the address of a and store in the pointer variable p
int b = *p;
Dereference p: Store in b the value of the variable that pointer variable p points to.
```
Following Up

For Functions in general:
“A Book on C”, Sections 5.1-5.6
(please ignore the comments on “traditional C” and C++)

For pointers:
“A Book on C”, Sections 6.1-6.3