Computer Programming: Skills & Concepts (CP1)
Parameters, & and ∗

14th October, 2010

Declaring functions, revisited

- functions must be declared before use
- but then low-level functions must come before high-level
- and the main program must come last

But we (most of us) find it easier to read programs ‘top-down’: high-level structure first, then fiddly detail.

Declaring functions, revisited

- functions must be declared before use
- but then low-level functions must come before high-level
- and the main program must come last

But we (most of us) find it easier to read programs 'top-down': high-level structure first, then fiddly detail.

There's a way to get round this:

- compiler only needs the function *header* to check it's correctly used;
- so declare the header first, then define the function later (e.g. after main program).

The disembodied header is called a function *prototype*.

This style of programming is widespread in Kelley and Pohl.

All the header files like `stdlib.h` and `descartes.h` contain prototypes, not code.

An example

```c
#include <stdlib.h>

int succ(int n);

int main(void) {
  printf("The successor of 3 is %d.\n", succ(3));
  return EXIT_SUCCESS;
}

int succ(int n) {
    return n+1;
}
```

Identifiers are optional in prototypes

```
#include <stdlib.h>

int succ(int);

int main(void) {
  printf("The successor of 3 is %d.\n", succ(3));
  return EXIT_SUCCESS;
}

int succ(int n) {
    return n+1;
}
```

A closer look at parameters

When a function (e.g. `int succ(int n)`) is called (e.g. `succ(a+2)`), what is the relation between the formal parameter `n` and the actual parameter `a+2` ?

There are several possible answers. Some (few) programming languages offer more than one. In C, there is just one: *call by value*.

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

Call by value

Again, consider `int succ(int n)` being called by `succ(a+2)`.
The actual parameter is an expression of a certain type (int in this case).
The formal parameter is a variable of the same type.

Call by value

Again, consider `int succ(int n)` being called by `succ(a+2)`.
The actual parameter is an expression of a certain type (int in this case).
The formal parameter is a variable of the same type.

- The actual parameter is evaluated to yield a value of the specified type. (Whatever the value of a+2 is.)

Call by value

Again, consider `int succ(int n)` being called by `succ(a+2)`.
The actual parameter is an expression of a certain type (int in this case).
The formal parameter is a variable of the same type.

- The actual parameter is evaluated to yield a value of the specified type. (Whatever the value of a+2 is.)
- The formal parameter is initialised to that value. (Recall that the formal parameter is a local variable of the function body.)

Call by value

Again, consider `int succ(int n)` being called by `succ(a+2)`.
The actual parameter is an expression of a certain type (int in this case).
The formal parameter is a variable of the same type.

- The actual parameter is evaluated to yield a value of the specified type. (Whatever the value of `a+2` is.)
- The formal parameter is initialised to that value. (Recall that the formal parameter is a local variable of the function body.)
- The function body is executed.

Call by value

Again, consider `int succ(int n)` being called by `succ(a+2)`.
The actual parameter is an expression of a certain type (int in this case).
The formal parameter is a variable of the same type.

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

Call by value

Again, consider `int succ(int n)` being called by `succ(a+2)`.
The actual parameter is an expression of a certain type (int in this case).
The formal parameter is a variable of the same type.

- The actual parameter is evaluated to yield a value of the specified type. (Whatever the value of `a+2` is.)
- The formal parameter is initialised to that value. (Recall that the formal parameter is a local variable of the function body.)
- The function body is executed.
- When `return` is reached (or the end of the function body) 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.

An example

```
int i = 3;

int succ(int n) {
  n = n+1;
  printf("Hi from \"succ\"!  The value of i is %d.\n", i);
  return n;
}

int main(void) {
  printf("The successor of %d is %d.\n", i, succ(i));
  printf("Hi from \"main\"!  The value of i is %d.\n", i);
  return EXIT_SUCCESS;
}
```

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?

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?

```
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;
}
```

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?

```
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*.

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.

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` is (essentially) the variable with that address.

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` is (essentially) the variable with that address.

And we can store addresses in variables (of type `int *`). So after

```
int x;
int * a = &x;
```

assigning to `*a` is the same as assigning to `x` and evaluating `*a` is the same as evaluating `x`.

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` is (essentially) the variable with that address.

And we can store addresses in variables (of type `int *`). So after

```
int x;
int * a = &x;
```

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 *`, but we're stuck with this insanity. There was a justification of sorts.

C programmers usually write `int *a;` rather than `int * a;`

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` is (essentially) the variable with that address.

And we can store addresses in variables (of type `int *`). So after

```
int x;
int * a = &x;
```

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 *`, but we're stuck with this insanity. There was a justification of sorts.
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 *`, `point_t *` and so on.

Swapping variables with & and *

```
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.

An example: ReadNumber from Practical 2

```
/*
 * Read a number from the input stream.
 *
 * value: On success, value receives the value read.
 *
 * Return - TRUE if successful, FALSE otherwise.
 */

int ReadNumber(int *value);
int ReadNumber(int *value) {
  int ch, total;

  ch = ReadSymbol();

  if (ch >= ’0’ && ch <= ’9’) {
    total = ch - ’0’;
    *value = total;
    return TRUE;
  } else {
    UnReadSymbol(ch);
    return FALSE;
  }
}
```

ReadNumber (continued)

```
int main(void) {
  int x;

  printf("Enter a number: ");
  if (!ReadNumber(&x)) {
    ParseError("Number Expected");
    return EXIT_FAILURE;
  }
  /* x now contains the number just read */
  printf("\nx = %d\n", x);
  return EXIT_SUCCESS;
}
```

Overview: Uses of & and *

```
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.