Functions

Lecture 4
CS 113 – Fall 2007

Announcements

- Add/drop deadline: Today!
- Assignment 1 due next Wednesday
  - Check newsgroup for clarifications, corrections, etc.
  - Need a partner? Check newsgroup.
- C compiler options
  - Eclipse + gcc
  - Alternative: Dev-C++
  - Another alternative: Turbo C
  - Xcode on Macs, including those in CIT labs.

A note on 113 assignments

- Please write clear, correct code
  - meaningful variable and function names
  - helpful comments
- Goal of assignments is to practice writing C programs
  - Unlike other CS courses, where more emphasis is on theory
  - Feel free to explore and use C language features, even ones we haven't covered in class
  - You can implement extra things not required by assignment

Order of evaluation

- *Operator precedence* and *associativity* rules define the order in which operators are evaluated
  - Some examples:
    - 5 + 3 / 2 = 5 + (3/2)
    - 1 - 1 - 1 = (1 - 1) - 1
    - 3 < 5 + 2 = 3 < (5 + 2)

Class	Associativity	Operators
Select	L→R	(...) [...] -> .
Unary	R→L	! ~ + - * & (*type*) sizeof ++ --
Binary arithmetical	L→R	* / %
Binary arithmetical	L→R	+ -
Shift	L→R	<< >>
Comparison	L→R	< <= > >=
Comparison	L→R	== !=
Binary bitwise	L→R	&
Binary bitwise	L→R	^
Binary bitwise	L→R	|
Binary boolean	L→R	&&
Binary boolean	L→R	||
Ternary	R→L	?...:
Assignments	R→L	= += -= *= /= &= != ^= <<= >>=
Sequence	L→R	,

Avoid confusing expressions

- Use parentheses to make precedence clear
  - Q: What does this code do?

```
void main()
{
  int a = -2, b = -1, c = 0;
  if( a < b < c )
    printf( "True.\n" );
  else
    printf( "False.\n" );

  if (a >= b >= c)
    printf( "True.\n");
  else
    printf( "False.\n");
}
```


Functions

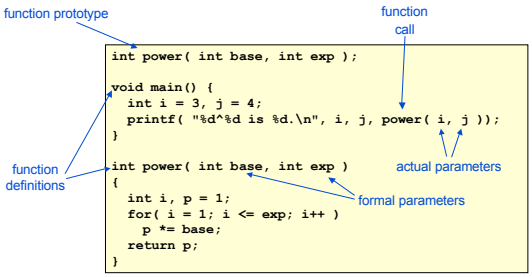
- Purpose of functions
  - Breaks a program into pieces that are easier to understand
  - Makes *recursive algorithms* possible to implement
  - Promotes *code reuse*
- Disadvantage of functions
  - Function calls add some memory and time overhead
- Functions in C
  - Similar to methods in Java
  - But C functions do not belong to a class. Every function is visible everywhere in the program.

A simple function

```
int power( int base, int exp )
{
  int i, p = 1;
  for( i = 1; i <= exp; i++ )
    p *= base;
  return p;
}
```

- This function takes 2 integers as arguments, returns an int
- Variables declared inside power (base, exp, p) are local variables and not visible outside the function
- `return` statement is used to specify return value

Simple function in context




```
int power( int base, int exp );

void main() {
  int i = 3, j = 4;
  printf( "%d^%d is %d.\n", i, j, power( i, j ));
}

int power( int base, int exp )
{
  int i, p = 1;
  for( i = 1; i <= exp; i++ )
    p *= base;
  return p;
}
```


Function return values

- If a function returns type `void`, then no `return` statement is needed
- If a function returns another type, then a `return` statement is required along all possible execution paths
  - What does this code do?

```
int foo( int arg );

int foo( int arg )
{
  if(arg == 1) return 0;
}

void main() {
  printf( "%d\n", foo(0) );
}
```


Call by value

- Function arguments in C are passed *by value*
  - The *value* of the argument is passed, not a reference
  - Functions are given a new copy of their arguments
  - So a function can't modify the value of a variable in the calling function (unless you use pointers)

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

void main() {
  int A = 1, B = 2;
  swap(A, B);
  printf( "%d %d\n", A, B );  // prints "1 2"
}
```


Call by value

- Call by value has advantages and disadvantages
  - Advantage: some functions are easier to write

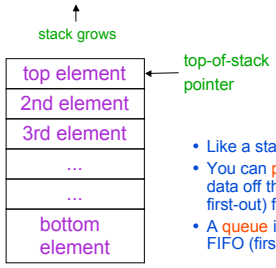
```
int power(int base, int exp)
{
  int result = 1;

  for( ; exp >= 1 ; exp--)
    result *= base;
  return base;
}
```

  - Disadvantage: sometimes you'd like to modify an argument (e.g. a `swap`() function)
    - We'll see how to do this using pointers later

Activation records

- When a function calls occurs, the function needs:
  - Its own copy of the actual parameters
  - Memory to store its local variables
  - An address to resume executing once the program returns
- C uses *activation records* to manage this information
  - The activation records are stored on the *execution stack*
  - Whenever a function is called, an activation record is created and *pushed* onto the stack
  - Whenever a function returns, its activation record is *popped* from the stack
  - The AR for main is always at the bottom of the stack

Stacks



- Like a stack of plates
- You can push data on top or pop data off the top in a LIFO (last-in-first-out) fashion
- A queue is similar, except it is FIFO (first-in-first-out)

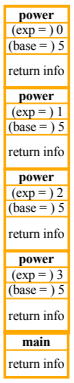
Recursion

- *Recursion* is a powerful technique for specifying sets, functions, and programs
  - A *recursive function* is a function that refers to itself
  - Many algorithms can be written more compactly, easily, elegantly using recursion
- Early languages (e.g. Fortran) did not support recursion
  - C and most other modern languages do
  - Recursion is possible because of the execution stack

Recursion example

- Example: computing positive integral powers
  - $a^n = a \cdot a \cdot a \cdots a$ (n times)
  - Recursive definition:
    - $a^0 = 1$
    - $a^{n+1} = a \cdot a^n$

```
int power(int base, int exp);

int power(int base, int exp)
{
  if(exp == 0) return 1;
  else return base*power(base, exp-1);
}

int main(void)
{
  printf("%d\n", power(5, 2));
  return 0;
}
```

Example: power(5, 2)

```
int power(int base, int exp);

int power(int base, int exp) {
  if(exp == 0) return 1;
  else return base*power(base, exp-1);
}

int main(void) {
  printf("%d\n", power(5, 2));
  return 0;
}
```


How Do We Keep Track?

- At any point in execution, many invocations of *power* may be in existence
  - Many stack frames (all for *power*) may be in Stack
  - Thus there may be different versions of the variables *base* and *exp*
- How does processor know which location is relevant at a given point in the computation?
  - Frame Base Register
    - When a method is invoked, a frame is created for that method invocation, and FBR is set to point to that frame
    - When the invocation returns, FBR is restored to what it was before the invocation
- How does machine know what value to restore in the FBR?
  - This is part of the return info in the stack frame

An aside: Loops in C

- So far we've seen 4 ways of writing loops:
  - while statements
  - do-while statements
  - for statements
  - recursion
- There's one more way…

goto statements

- A `goto` statement explicitly forces execution to jump from one part of the program to another
  - Syntax: `goto label`
  - One application is looping, e.g.:

```
int main(void)
{
  int j=0;

begin_loop:
  printf("%d\n", j);
  j++;
  if(j < 10) goto begin_loop;

  return 0;
}
```


A common use of goto

- `goto` statements are often used to handle errors
  - like a rudimentary form of exception handling in Java or C++

```
int function(void)
{
  int j=0;

  for( . . . )
   for( . . . )
    while( . . . ) {
       /*lots of complex code */
       if(error) goto abort;
       /* more code */
    }

abort:
  printf("error detected!\n")
  return 0;
}
```


Use of goto

"GOTO is a four letter word." -- Edsger Dijkstra

- gotos should almost always be avoided
  - goto statements make code harder to understand
  - It's always possible to rewrite code without a goto
- goto statements have become nearly extinct
  - They were very common in older languages
  - Modern programming constructs (`for` loops, etc.) make them unnecessary
  - Many newer languages (e.g. Java) don't support them at all

Multiple source files

- C lets you break up a program into several source files
  - Unlike Java or Matlab, it doesn't *require* you to do this
  - How you break up the program is up to you
- To do this,
  - Put the code for a function in exactly one file
  - Put a function prototype in any file that uses the function

Example: multiple source files

main.c

```
#include <stdio.h>

int power(int base, int exp);
int cube(int base);

int main(void)
{
  printf("%d\n", power(5, 3));
  printf("%d\n", cube(10));
  return 0;
}
```

power.c

```
int power( int base, int exp )
{
  int i, p = 1;
  for( i = 1; i <= exp; i++ )
    p *= base;
  return p;
}
```

cube.c

```
int power(int base, int exp);

int cube(intbase)
{
  return power(base, 3);
}
```