Functions Calls

Davood Rafiei

Predefined Functions

- Libraries
  - Check the libraries before implementing your own function
  - Must '#include' appropriate libraries
- Two types of functions:
  - Those that return a value
  - Those that do not (void)

Predefined Functions (2)

- Math functions very plentiful
  - Found in library <math.h>
  - Most return a value (the ‘answer’)
- Examples
  - *double sin(double x)*
  - *double asin(double x)*
  - *double log10(double x)*
  - *double pow(double x, double y)*
  - *double round(double x)*
  - *double floor(double x)*
  - *double ceil(double x)*
  - *int abs(int i)*
  - *int rand()*
- And many more

Predefined Functions (3)

- Character handling: #include <ctype.h>
  - E.g.
    - *int isalpha(int c)*
    - *int islower(int c)*
    - *int isdigit(int c)*
- String handling: #include <string.h>
  - E.g.
    - *memcpy, strcpy, strcat, strcmp, strchr, strstr, …*

Programmer-Defined Functions

- Write your own functions
- Task break down
  - Recall the idea of “each program does one thing”
  - Divide & conquer
- File break down
  - Group related functions into a file

Function Definition

- Functions are ‘equals’; no function definition is ever ‘part’ of another (in c99).
- A function definition includes
  - Function prototype
    - Parameter type(s)
    - Return type
  - Body implementation
    - Uses variable names to refer to parameters
    - Has a return statement (if return type is not void)

Function Prototype

- Used for type checking (by the compiler)
  - argument types
  - return type
- Must be declared before an invocation
- Examples

```
void parseFile(FILE * fp, char * fname);
FILE *popen(const char *command, const char *type);
void bufferFile( FILE *, struct aLine *, int *, int);
void classifyLine( struct aLine *, int);
void printClass( struct aLine *, int);
```

Function Prototype (2)

- Good or bad?
  - *void init();*
  - *init();*
  - *void init(void);*
  - *int* busses(char *address);*
  - *double area(double length, width);*

Function Prototype (3)

- #include files typically contain a lot of function prototypes.
  - *#include <math.h>*

- Don’t forget ‘;’ at the end of function prototype
  - *double mySqrt(double x);*

- Can the return type be an ‘array’?
  - It can be a pointer (to an array)

Call by Value/Reference

- Call by value
  - Copy of data passed to function
  - Changes to copy do not change original
  - Used to prevent unwanted side effects
- Call by reference
  - Function can directly access data
  - Changes affect original

Example: square

```
double squareByValue(double x) {
  return x*x;
}
void squareByRef(double *x) {
  *x = (*x)*(*x);
}
int main() {
  double v = squareByValue(3.0);
  squareByRef(&v);
  printf(“%f\n”, v);
}
```

Example: square (2)

- `double d=2.0;`
- `int i=1;`
- `squareByVal(i+4);`
- `squareByVal(squareByVal(d+1));`
- `squareByRef(&d);`
- `squareByRef(d+4.5);  //illigal!`
- `squareByRef(&i);`

Example: Swap

```
void swapV(int x, int y)
{
  int tmp = x;
  x = y;
  y = tmp;
}

void swapR(int *x, int *y)
{
  int tmp = *x;
  *x = *y;
  *y = tmp;
}
```

Example: Swap (2)

```
int main()
{
  int a = 12, b = 9;
  printf("Before swapV: a= %d, b=%d\n", a, b);
  swapV(a, b);
  printf("After swapV: a= %d, b=%d\n", a, b);

  int c = 7, d = 5;
  printf("Before swapR: c= %d, d=%d\n", c, d);
  swapR(&c, &d);
  printf("After swapR: c= %d, d=%d\n", c, d);
}
```

Example: Swap (3)

```
drafiei@ug20:~/201>gcc -Wall -std=c99 swap-val-ref.c
drafiei@ug20:~/201>./a.out
Before swapV: a= 12, b=9
After swapV: a= 12, b=9
Before swapR: c= 7, d=5
After swapR: c= 5, d=7
```

“Call by Value” vs. “Call by Reference”

- Call by value
  - Generally preferred since there is less dependency between the caller and the callee.
- Call by reference
  - Useful when
    - the function has multiple outputs.
    - the state of the argument need to be altered.
    - the argument is a large object.

Type Promotion

```
#include <stdio.h>
int  squareI(int n) {return n*n;}
float  squareF(float n) {return n*n;}
double  squareD(double n) {return n*n;}

int main()
{
  int i=10; float f=2.5; doube d=3.5;
  printf(“squareI(%d): %d\n”, i, squareI(i));
  printf(“squareF(%d): %f\n”, i, squareF(i));
  printf(“squareD(%d): %f\n”, i, squareD(i));
  printf(“squareD(%f): %f\n”, f, squareD(f));
  printf(“squareF(%f): %f\n”, d, squareF(d));
  printf(“squareI(%f): %d\n”, d, squareI(d));
  return 0;
}
```

Type Promotion (2)

```
drafiei@ug20:~/201>gcc -Wall -std=c99 square-conversion.c
drafiei@ug20:~/201>./a.out
squareI(10): 100
squareF(10): 100.000000
squareD(10): 100.000000
squareD(2.500000): 6.250000
squareF(3.500000): 12.250000
squareI(3.500000): 9
```

Variable Number of Arguments

- Let’s revisit *scanf* and *printf*
  - *scanf(“format”, a1, a2, a3, a4);*
  - *printf(“format”, b1, b2);*
- With user-defined functions
  - Can do the same

```
drafiei@ug20:~/201>cat var-args.c
#include <stdio.h>
#include <stdarg.h>

void printIntArg(int n, ...) {
  int arg;

  va_list ap;
  va_start(ap, n);            // make ap point to 1st unnnamed argument
  for (int i=0; i<n; i++) {
    arg = va_arg(ap, int);  //return one argument of type int and step ap to the next
    printf("Arg %d: %d\n", i, arg);
  }
  va_end(ap);  // cleanup
}

int main()
{
  printf(“--1st call\n”);
  printIntArg(2, 10, 20, 30);
  printf(“--2nd call\n”);
  printIntArg(1, 10, 20, 30);
  printf(“--3rd call\n”);
  printIntArg(5, 10, 20, 31, 40, 50);
}
```

```
drafiei@ug20:~/201>gcc -Wall -std=c99 var-args.c
drafiei@ug20:~/201/c-samples>./a.out
--1st call
Arg 0: 10
Arg 1: 20
--2nd call
Arg 0: 10
--3rd call
Arg 0: 10
Arg 1: 20
Arg 2: 31
Arg 3: 40
Arg 4: 50
```

Not Covered

- Storage class of a function
  - For example, to limit the accesses
- Recursion
  - Generally not recommended!
  - See examples in the textbooks (both K and KR)