COMP 208
Computers in Engineering

Lecture 15

Jun Wang
School of Computer Science
McGill University

Fall 2007

Review

- Compound assignment: `var op= exp;`

- The if statement

```
if (expression)
  statement1
else
  statement2
```

- Assignment statement in C has value

```
if ((a = b+c) > d)
  f++;
```

- Logical value: in C, the value 0 is used to represent logical FALSE, and any non-zero value can be used to represent logical TRUE

Review

- Loops: for-loop, while-loop, do-while

```
for (e1; e2; e3)
   statement
```

```
while (expression)
   statement
```

```
do
   statement
while (expression)
```

scanf and printf Format Codes

- Syntax:
  - scanf(<formats>, <list of variables>);
  - printf(<formats>, <list of variables>);
- Formats:
  - d: decimal int
  - o: octal int
  - x: hexdecimal int
  - c: character
  - s: string
  - f: real number, floating point
  - e: real number, exponential format

Escape Sequences

- Some escape sequences:

Escape Sequence	**Meaning**
`\b`	backspace
`\t`	tab
`\n`	newline
`\r`	carriage return
`\"`	double quote
`\'`	single quote
`\\`	backslash

Arrays in C

One dimensional arrays are defined as:

```
<type> name[size];
```

Array subscripts start at 0 and end one less than the array size.

For example the array

```
int list[50];
```

is an array of 50 integer values indexed from 0 to 49

In Fortran:

```
INTEGER :: list(50)
```

Arrays in C

Accessing individual components is done by indexing.

This is similar to Fortran, except the syntax uses square brackets.

```
Thirdnumber = list[2];   //read
list[5] = 100*list[3];   //write
```

Multi-Dimensional Arrays

Multi-dimensional arrays are defined as:

```
int tableofnumbers[50][50];
```

For more dimensions add more [ ]:

```
int bigD[50][50][40][30]...[50];
```

Elements are accessed as:

```
anumber = tableofnumbers[2][3];
tableofnumbers[25][16]=100;
```

Character Strings

C Strings are defined as arrays of characters.

```
char name[50];
```

C has no string handling facilities built in and so the following are all **illegal**:

```
char first[50],last[50],full[100];
first="Arnold"; /* Illegal */
last="Schwarznegger"; /* Illegal */
full="Mr"+firstname+lastname;
  /*illegal*/
```

Strings

There is a special library of string handling routines in **`<string.h>`**

To print a string use printf with a **%s** control:

```
printf(“%s”,name);
```

In order to allow variable length strings the \0 character is used to indicate the end of a string.

In C, a string literal is automatically appended a ‘\0’ character. e.g. “Nathan” is actually “Nathan\0”.

If we have a string, `char name[50];` we can store the string “Nathan\0” in it.

String and `char` array

```
char s[50];

s[0] = ‘a’;
s[1] = ‘b’;
s[2] = ‘c’;
s[3] = ‘\0’;

printf(“%s\n”, s);  // prints “abc”
```

- A char literal is enclosed in `‘’`; only one character allowed: `‘a’`, `‘x’`, `‘\0’`, `‘\n’`
- A string literal is enclosed in `“”`; A null character `‘\0’` is automatically appended: `“abc”`, `“hello world”`, `“x”`
- A char array can be used to store a string; but you must have a ‘\0’ at the end

Functions

Syntax of Function Definitions:

```
   returntype name (parameter list)
    {
      localvariable declarations
      functioncode
     }
```

The parameter list is a list of names together with the associated type

Function Example

```
float findaverage(float a, float b)
{
        float average;
        average=(a+b)/2;
        return average;
}
```

Using Function

```
float findaverage(float a, float b) {
      float average;
      average=(a+b)/2;
      return average;
}
```

We *use* the function as follows:

```
main() {
 float a=5,b=15,result;
 result = findaverage(a,b);
 printf(“average=%f\n”,result);
}
```

Void Functions

If you do not want to return a value use the return type void:

```
void squares()
{
  int loop;
  for (loop=1; loop<10; loop++)
     printf(“%d\n”, loop*loop);
}
main()
{
  squares();
}
```

Array Parameters

Single dimensional arrays can be passed to functions as follows:-

```
float findaverage(int size,float list[])
{
  int i;
  float sum=0.0;
  for (i=0;i<size;i++)
     sum += list[i];
  return sum/size ;
}
```

Note we do not specify the dimension of the array when it is a *parameter* of a function.

Multi-Dimensional Arrays

```
void print_table(int xsize, int ysize,
                 float table[][5])
{
   int x,y;
   for (x=0;x<xsize;x++) {
      for (y=0;y<ysize;y++)
         printf(“\t%f",table[x][y]);
      printf(“\n");
  }
}
```

Note we must specify the second (and subsequent) dimensions of the array but not the first dimension.

Function Prototypes

Before using a function C must *know* the type it returns and the parameter types.

Function prototypes allow us to specify this information before actually defining the function

This allows more structured and therefore easier to read code.

It also allows the C compiler to check the *syntax* of function calls.

Function Prototypes

If a function has been defined before it is used then you can just use the function.

If NOT then you must *declare* the function prototype. The prototype declaration simply states the type the function returns and the type of parameters used by the function.

Function Prototypes

It is good practice to prototype all functions at the start of the program, although this is not strictly necessary.

Another way, which is very common with C programs, is to put the prototypes in a header file.

Function Prototypes

A function prototype has

1. the type the function returns,
2. the function name and
3. a list of parameter types in brackets

e.g.

```
int strlen(char []);
```

This declares that a function called `strlen` returns an integer value and accepts a string as a parameter.

Coercion or Type-Casting

Mixed mode operations are handled by C very much like Fortran handles them.

Integer values are converted to reals when assigning to a real and when performing an arithmetic operation using integers an reals

Floating point numbers are truncated to integers when assigning then to an integer variable

In C however, the programmer is able to control this using a cast operator () to force the coercion of one type into another

Coercion or Type-Casting

```
int integernumber;
float floatnumber=9.87;
integernumber = (int)floatnumber;
```

assigns 9 (the result is truncated) to integernumber.

```
int integernumber=10;
float floatnumber;
floatnumber=(float)integernumber;
```

assigns 10.0 to float number.

None of the 2 castings above are necessary,
but they make the programmer’s intention clear.

Type Coercion

Coercion can be used with any of the simple data types including char, so:

```
int integernumber;
char letter=‘A’;
integernumber=(int)letter;
```

casting not necessary here

assigns 65 (the ASCII code for ‘A’) to integernumber.

Coercion

Another use is to make sure division behaves as requested:

To divide two integers intnumber and anotherint and get a float

```
floatnumber =
    (float)intnumber /(float)anotherint;
```

ensures floating point division.

```
r = (double) rand() / (RAND_MAX + 1.0);
```

Global Variables

Variables defined **outside functions** are global variables, available to any code following their definition.

Thy can be initialized when declared, or initialized in `main()`

Global Variable Example

```
int Called = 0;

void foo()
{
    Called++;
    // blah
}
void bar()
{
    Called++;
    foo();
    // blah
}

int main()
{
    foo();
    foo();
    bar();
    printf(“function calls made: %d\n”, Called);
}
```

They went that away!

Pointers in C

Nathan Friedman

Fall, 2007

Swap Two Values

```
void swap(int x, int y)
 { int temp;
   temp = x;
   x = y;
   y = temp;
 }
```

Swap Two Values

```
void swap(int x, int y)
 { int temp;
   temp = x;
   x = y;
   y = temp;
 }

void main ()
{
      int a, b;
      a = 27;
      b = 103;
      swap(a,b);
      printf ("%d   %d \n", a, b);
}
```

Surprise!

```
>swap
27    103
```

We wanted to see:

```
103    27
```

What happened here?

This worked in Fortran!

Why not in C?

Parameter Passing

- It turns out that Fortran and C handle parameters very differently
- In C all parameters are passed by value
- The parameters are treated as new local variables that are initialized to the argument values
- Any changes made are local and do not effect the arguments

Fortran ?

- Remember how arguments were passed in Fortran?
  - Expressions or constants had their values put in new local variables representing the parameters (**call by value**)
  - If the argument was a variable the parameter was treated as an alias for that variable (**call by reference**)

C ?

- C is more uniform
- It treats all arguments the same way, the way Fortran treats expressions
- But ...

  That means that we have problems with functions like swap where we want the argument values to change

What’s the solution?

- C allows us to manipulate addresses, called pointers
- If we pass a pointer as an argument, the value of the argument doesn’t change
- But . . .

  The value in the cell pointed to could change

Swapping Values in C

```
void swap(int *px, int *py)
 { int temp;

   temp = *px;
   *px = *py;
   *py = temp;
}
```

Let's Compare

```
void swap(int x, int y)
 { int temp;

   temp = x;
   x = y;
   y = temp;
 }
```

```
void swap(int *px, int *py)
 { int temp;

   temp = *px;
   *px = *py;
   *py = temp;
 }
```

What is a Pointer?

A pointer is a variable which contains the memory address of another variable.

Declaring a Pointer

To declare a pointer to an integer variable:

```
int *ip;    //or  int* ip;
```

The variable ip will be able to store the **address** of an integer cell

In general:

```
type *name;
```

Type

We can have a pointer to any variable type (cell “shape”)

Once we declare it, the pointer can only be associated with a variable of the specific type we declared

How do we get the address of a variable?

Suppose we have an integer variable `x` that contains some value, say 37.

If we want the address of `x`, we can use the & operator.

For example we could write

```
ip = &x;
```

That is, `ip` contains the address of `x` and `*ip` has the value 37

How do we find the value pointed to?

If the variable ip contains an address, how do we find out what is stored in the cell pointed to?

We use a dereferencing operator.

The dereference operator * returns the "contents of the object pointed *to*"

A Simple Example

```
int x = 1, y = 2;
int *ip;

ip = &x;    /* ip gets the address of x */
y = *ip;    /* y gets assigned 1 */
*ip = 3;    /* x gets assigned 3 */
```

Note that `*ip` can appear both on the left-hand side of an assignment, and the right-hand side.

Swapping Values in C

```
void swap(int *px, int *py)
 { int temp;

   temp = *px;
   *px = *py;
   *py = temp;
}

void main () {
   int a, b;
   a = 27;
   b = 103;
   swap(&a,&b);
   printf ("%d   %d \n", a, b);
}
```

Swapping Values in C

```
void swap(int *px, int *py)
 { int temp;

    temp = *px;
    *px = *py;
    *py = temp;
}

void main () {
   int a, b;
    int *pa = &a, *pb = &b;
   a = 27;
   b = 103;
   swap(pa, pb);
   printf ("%d   %d \n", a, b);
}
```

demo