NWEN 241
C Control Constructs and Functions

Winston Seah

School of Engineering and Computer Science
Victoria University of Wellington

Built-in Operators

- arithmetic
- increment/decrement
- assignment
- relational
- logical
- bitwise
- others including type casting

- Pointers related operators (*, &, ->)

Arithmetic operators

\+ - * / %

Note:

(i) ***a*** % ***b*** ***a*** modulus ***b***

which means *remainder* after dividing ***a*** by ***b***.

(ii) printf("%d", a);

conversion specification / format

Increment/Decrement operators

++ --

Can be applied to variables, but not constants and ordinary expressions;

```
int  a, b, c = 0;     /* declare and initialize all to 0 */

a = ++c; /* prefix operator; increment c, then assign to a */
b = c++; /* postfix operator; assign to b, then increment c */
printf("%d %d %d\n", a, b, ++c);
```

Output?

Assignment Operators (1)

To change the value of a *variable,* e.g. a = 2

Assignment expression

```
a = b + c;
```

Assignment statement

Value of right side ‘b + c’ is assigned to left side ‘a’, and that value also becomes the value of the expression.

```
b = 4;
c = 2;
a = b + c;
```

} a = (b = 4) + (c = 2);

Assignment Operators (2)

More assignment operator combinations, e.g.

a = a + 3	a **+=** 3
b = b * (c + 3)	b *= (c + 3)

How do you condense b = b * c + 3 ?

(b *= c) + 3

All assignment operators:

= += -= *= /= %=

>>= <<= &= ^= !=

Data Input and Output

Functions for data input and output

- getchar() / putchar()

```
char c;
c = getchar();      /* input a char */
putchar(c);         /* output a char */
```

- gets() / puts()

```
char line[80];
gets(line);         /* input a line/string */
puts(line);         /* output a line */
```

- **scanf() / printf()**

Data Input and Output

scanf() / printf()

```
int i;
float f;
char c;
char s[80];
scanf("%d", &i);            /* %d is format information
                             * d is conversion character
                             */
scanf("%f", &f);            /* &f is f's memory address
                             * input is sent to &f
                             */
```

Data Input and Output

scanf() / printf()

```
int i;
float f;
char c;
char s[80];
scanf("%d", &i);            /* %d is format information
                             * d is conversion character
                             */
scanf("%f", &f);            /* &f is f's memory address
                             * input is sent to &f
                             */
printf("\nYou typed in \"%f\"\n", f);
                            /* \n starts new line. \" treats "
                             * as an ordinary character
                             */
```

Data Input and Output

scanf() / printf()

```
int i;
float f;
char c;
char s[80];
scanf("%d", &i);            /* %d is format information
                             * d is conversion character
                             */
scanf("%f", &f);            /* &f is f's memory address
                             * input is sent to &f
                             */
printf("\nYou typed in \"%f\"\n", f);
                            /* \n starts new line. \" treats "
                             * as an ordinary character
                             */
scanf(" %c", &c);           /* blank space preceding %c to
                             * ignore \n typed in earlier
                             */
scanf("%s", s);             /* a seq. of nonwhite space char */
```

Control Constructs

Loops: for, while and do-while

```
#include <stdio.h>     /* each loop runs 4 times */

int main(void)
{ int i = 0, x =0;
  for (; i <4; i++)    /* starting and ending conditions */
  { x += i;
    printf("for loop: x = %d, i = %d\n", x, i);
  }

  while (i < 2*4)      /* only given ending condition */
  { x += i;
    printf("while loop: x = %d, i = %d\n", x, i);
    i++;
  }

  do                   /* do at least once */
  { x += i;
    printf("do-while loop: x = %d, i = %d\n", x, i);
    i++;
  } while (i < 3*4);   /* ending condition */
  return 0;
}
```

Control Constructs

Blocks

```
int main(void)
{ int i = 0, x =0;



  for (int i=-4; i < 4; i++)  /* i is re-declared. */
  { x += i;
  }

  while (i < 2*4)
  { x += i;
    i++;
  }

  do
  { x += i;
    i++;
  } while (i < 3*4);

  return 0;
}
```

Control Constructs

Blocks

```
int main(void)
{ int i = 0, x =0;                  /* i will be used by the */
                                    /* 'while' and 'do-while' loops, */
                                    /* but not the 'for' loop */

  for (int i=-4; i < 4; i++)        /* i is re-declared. */
  { x += i;                         /* only valid within this block. */
  }

  while (i < 2*4)                   /* The 2nd i has no effects */
  { x += i;                         /* in this and next block */
    i++;
  }

  do
  { x += i;
    i++;
  } while (i < 3*4);

  return 0;
}
```

Control Constructs

Conditionals: if-else and switch

```
int main(void)            /* to test if it is an upper-case alphabetic letter */
{ char i, c;
  printf("\nPlease enter an alphabetic character:\n");
  c = getchar();




}
```

Control Constructs

Conditionals: if-else and switch

```
int main(void)              /* to test if it is an upper-case alphabetic letter */
{ char i, c;
  printf("\nPlease enter an alphabetic character:\n");
  c = getchar();

  if (isalpha(c))           /* true = nonzero, false = zero */
    ;                       /* empty is ok, but ";" must be there */
  else
    return(printf("You did not enter an alphabetic character\n"));
}
```

Control Constructs

Conditionals: if-else and switch

```
int main(void)              /* to test if it is an upper-case alphabetic letter */
{ char i, c;
  printf("\nPlease enter an alphabetic character:\n");
  c = getchar();

  if (isalpha(c))           /* true = nonzero, false = zero */
    ;                       /* empty is ok, but ";" must be there */
  else
    return(printf("You did not enter an alphabetic character\n"));

  if (isupper(c) ? 1 : 0)                    /* true = 1, false = 0 */
    printf("if-else: it is an upper-case letter\n");
  else
    printf("if-else: it is a lower-case letter\n");




}
```

Control Constructs

- Conditionals: if-else and switch

```
int main(void)              /* to test if it is an upper-case alphabetic letter */
{ char i, c;
  printf("\nPlease enter an alphabetic character:\n");
  c = getchar();

  if (isalpha(c))           /* true = nonzero, false = zero */
    ;                       /* empty is ok, but ";" must be there */
  else
    return(printf("You did not enter an alphabetic character\n"));

  if (isupper(c) ? 1 : 0)                    /* true = 1, false = 0 */
    printf("if-else: it is an upper-case letter\n");
  else
    printf("if-else: it is a lower-case letter\n");

  i = (isupper(c) != 0 ? 'T' : 'F');         /* true = 'T', false = 'F' */
  switch(i) {
  case 'T':
    printf("switch: it is an upper-case letter\n");
    break;                  /* break must be there, otherwise it will go through */
  case 'F':
    printf("switch: it is a lower-case letter\n");
  }
  return 0;
}
```

Control Constructs

- break, continue and goto
  - **break**: jumps out of the loop
  - **continue**: stops current iteration and starts next iteration
  - **goto** jumps to a labelled statement
  - Java support labelled **continue** and **break** statement
  - Java does not support **goto**

Functions in C Programs

- Every C program has at least one function: main()
- No C program **needs** to have more than one function in it
  - Everything can be put in main():

Functions in C Programs

- Every C program has at least one function: main()
- No C program **needs** to have more than one function in it
  - Everything can be put in main(): not a good idea
- Any C program with only a main function is almost certainly for training purposes
- What are functions good for?
  - structuring our thoughts (structured programming)
  - allowing us to re-use code, reducing work and reducing errors
- A C program can be modularised by functions
  - A big program can be broken down into a number of smaller ones

Creating a Simple Function

- Suppose we frequently wanted to compare two integers and then use the larger. We might have code like this repeatedly written in our program:

```
int p, q, l;
...                    /* p, q initialised */
if (p > q)
  l = p;
else l = q;
...                    /* l gets used */
```

Creating a Simple Function

- How about making it a stand-alone function?

```
l = larger(p, q);
```

- What we need to do:
  - Pick a name for the function: larger()
  - Specify what type of variables that larger() is going to compare:

Creating a Simple Function

- How about making it a stand-alone function?

```
l = larger(p, q);
```

- What we need to do:
  - Pick a name for the function: larger()
  - Specify what type of variables that larger() is going to compare: larger(int, int)
  - Specify what type of value that larger (int, int) is going to return:

Creating a Simple Function

- How about making it a stand-alone function?

```
l = larger(p, q);
```

- What we need to do:
  - Pick a name for the function: larger()
  - Specify what type of variables that larger() is going to compare: larger(int, int)
  - Specify what type of value that larger (int, int) is going to return: int larger(int, int)
  - **int larger(int, int)**: this is called function prototype / declaration

Creating a Simple Function

- How about making it a stand-alone function?

```
l = larger(p, q);
```

- What we need to do:
  - Pick a name for the function: larger()
  - Specify what type of variables that larger() is going to compare: larger(int, int)
  - Specify what type of value that larger (int, int) is going to return: int larger(int, int)
  - **int larger(int, int)**: this is called **function prototype** / declaration
- Code it: function definition/implementation

Creating a Simple Function

- Function definition

```
int larger(int x, int y)
{
  if (x > y)
    return x;
  else return y;
}
```

- x and y are called “formal parameters”, whose scope is the body of the function.

Creating a Simple Function

- Let us use larger()

```
...
int main(void)
{
  ...
  l = larger(p, q); /* p and q are called "actual */
  ...               /* parameters". Their values are */
}                   /* going to be copied to x and y. */

int larger(int x, int y)
{
  if (x > y)
    return x;
  else return y;
}
```

Creating a Simple Function

- Let us use larger()

```
...
int main(void)
{
  ...
  l = larger(p, q); /* p and q are called "actual */
  ...               /* parameters". Their values are */
}                   /* going to be copied to x and y. */

int larger(int x, int y)
{                   /* x and y (NOT p and q) are */
  if (x > y)        /* going to be compared here. */
    return x;       /* the larger value is going to be */
  else return y;    /* returned to larger(p, q). */
}
```

Creating a Simple Function

- Function prototype

```
int main(void)
{ ...
  l = larger(p, q);     /* larger() not declared yet */
  ...
}
int larger(int x, int y)
{ ...
}
```

  - This is not good …
  - Use function prototype to declare the function before being used

```
int larger(int, int);

int main(void)
{...}

int larger(int x, int y)
{...}
```

Creating a Simple Function

- Function call: `l = larger(p, q);`
- Pass by value
  - The values of “actual parameters” (p, q) are copied to “formal parameters” (x, y)
  - “actual parameters” and “formal parameters” are separate entities
  - What happens thereafter to “formal parameters” has nothing to do with “actual parameters”
    - Any changes on x, y will not be transferred back to p, q

Another Simple Function

- Swap the values of two variables:

```
...             /* p, q, tmp declared */
...             /* p, q initialised */
tmp = p;
p = q;
q = tmp;
```

- Turn this into a function.
  - Define the data types

Another Simple Function

- A function for swapping
  - The function does not return a value
  - What is the return type then: **void**

```
void swap(int, int); /*function prototype*/
void swap(int x, int y)
{
   int tmp;
   tmp = x;
   x = y;
   y = tmp;
}
```

Another Simple Function

- Does it work?

```
int main(void)
{ int p = 40;
  int q = 80;
  swap(p, q);      /* the values of p, q */
  return 0;        /* are copied to x, y */
}
void swap(int x, int y)
{ int tmp;
  tmp = x;
  x = y;
  y = tmp;
}
```

Another Simple Function

- Does it work?

```
int main(void)
{ int p = 40;
  int q = 80;
  swap(p, q);      /* the values of p, q */
  return 0;        /* are copied to x, y */
}
void swap(int x, int y)
{ int tmp;
  tmp = x;
  x = y;
  y = tmp;         /* x, y get swapped */
}
```

Another Simple Function

- Solution: pass in the *addresses* of p, q
  - &p is the address of the memory that stores p's value
  - The values of p, q are stored at &p, &q
  - We use *pointers* to store the addresses of p, q

```
int *ptrp, *ptrq;   /* declare pointers */
ptrp = &p; /* &p stored in ptrp */
ptrq = &q; /* &q stored in ptrq */
```

  - *ptrp, *ptrq give us access to the values stored at &p, &q

```
printf("p=%d; q=%d", *ptrp, *ptrq);
tmp = p;
*ptrp = q;   /* equivalent to p=q; */
*ptrq = tmp; /* p, q get swapped */
```

Another Simple Function

- The function

```
void swap(int *, int *);
void swap(int *ptrx, int *ptry)
{ int tmp;
  tmp = *ptrx;
  *ptrx = *ptry;
  *ptry = tmp;
}
```

Another Simple Function

- Let us do the swap

```
int main(void)
{ ...
  int *ptrp, *ptrq;
  ptrp = &p;
  ptrq = &q;
  swap(ptrp, ptrq); /*the addresses of p, q*/
  return 0;         /*are passed to swap() */
}
void swap(int *ptrx, int *ptry)
{ int tmp;
  tmp = *ptrx;
  *ptrx = *ptry;    /*the values stored at */
  *ptry = tmp;      /*the addresses of p, q*/
}                   /*are swapped */
```

Another Simple Function

- Pass by address emulating pass by reference
  - The values of “actual parameters” (ptrp, ptrq) are copied to “formal parameters” (ptrx, ptry)
  - The values are memory addresses
  - “actual parameters” and “formal parameters” hold the addresses of the same memory blocks
  - *ptrx, *ptry give you the access to the memory
    - Any changes on *ptrx, *ptry change the values stored in the memory
- More about pointers later…

Pass by value vs pass by address

- Are they the same?

- What are used in Java?
  - Pass by value?
  - Pass … by value?

Pass by value vs pass by address

- Are they the same?
  - Yes, addresses are values
- What are used in Java?
  - Pass by value: primitive types
  - Pass reference by value: objects
  - Pass reference by value is pass by value

- Pass by reference? (more later…)

Summary

- Operators
- Data input/output
- Control Constructs
- Why functions
- How to use functions
- A little bit about pointers