Overview

• Announcements
• Function Calls (Chapter 1.7-1.9, C programming language)
• Structs, union, typedef, bitfield, enum
  • Chapter 6.1-6.4, 6.7,6,8,6.9, and 2.3 (C programming language)
• Pointers (Chapter 5.1 -5.6, C programming language)
  • for, if, else, switch, while, etc.

Announcements

• Homework 2 is due Sunday (5/29 – midnight) Black Board
• This week:
  – Lab 2: Robot movement
  – Lab 3: Push Button, switch, stepper motor

Function Calls (short intro)

• Syntax is just like Java
• Parameters can be passed by
  – value
  – address (will cover in detail after introducing pointers)

Example of calling a function:
myFunction(param1, param2);

• Implicit Declaration warning – these occur if you try to
call a function that hasn’t been defined yet!

int add(int x, int y)
{
    return x + y;
}
Function Calls (short intro)

```c
int add(int x, int y)
{
    return x + y;
}

void main()
{
    int r = 5;
    r = add(3, 3);  // Warning - implicit declaration
    // r is now 6
}
```

Function Calls (short intro)

```c
int add(int x, int y) // best practice: add at top of file (prototype), // or include a header file
{
    return x + y;
}

void main()
{
    int r = 5;
    r = add(3, 3);  // Warning - implicit declaration
    // r is now 6
}
```

Reserved Words in C

- char
- double
- float
- int
- long
- short
- void
- enum
- struct
- union
- typedef
- break
- case
- continue
- default
- do
- else
- for
- goto
- if
- return
- switch
- while
- auto
- const
- extern
- register
- signed
- static
- unsigned
- volatile
- sizeof
enum

- Chapter 2.3: C programming Language

enum

- The enum type allow a programmer to define variable that may set to equal to a set of user defined names

```c
enum compass_direction{
    north, east, south, west
};

enum compass_direction my_direction;
my_direction = west;
```

struct

- Chapter 6.1 - 6.5 (C programming Language)

struct

- The struct type allows a programmer to define a compound data type.
- The size of a struct is the size of its components added together.

```c
struct RGB{
    char red;
    char green;
    char blue;
};

struct RGB my_color;
my_color.blue = 255;
struct RGB *my_color_ptr = &my_color;
(*my_color_ptr).blue = 255;
// equivalent to previous line
```

struct

```c
struct student {
    char name[30];
    int ISUID;
};

struct student student_records[100];
studentent_records[10].ISUID = 5678; // Set student at index 10 ISUID
```

Bit Fields in Structures

```c
struct KBCR {
    unsigned int model : 4;
    unsigned int KBERROR : 1;
    unsigned int CAPLOCK : 1;
    unsigned int READY : 1;
} KBCR;
```
Bit fields in structures

```c
if (KBCR.READY) {  
    struct KBCR *pKBCR;
    pKBCR->CAPLOCK = 0;
}
```

Bitfields

```c
struct MyBitField{
    char clockselect : 3;
    char clockenable : 1;
    char operationmode : 4;
};
```

Chapter 6.9: C-programing language

union

- Chapter 6.8: C programming language

union

```c
union u_tag {
    int ival;  // size two bytes
    float fval; // size four bytes
    char *sval; // size two bytes
};
```

The size of a union variable is the size of its maximum component.

Fall 2011

Union

- Maintain multiple views: byte or bit structure.

```c
union KBCR_U {
    struct KBCR KBCR;
    uint8_t KBCR_Aggregate;
} KBCR_U;
```

```c
KBCR_U.KBCR.READY = 1;
```

Fall 2011
Structure and Union

Use of union inside of a struct

```c
struct {
    char *name;
    int flags;
    int utype;
    union {
        int ival;
        float fval;
        char *sval;
    } u;
} symtab;
```

How large is the struct `symtab`?

```
Fall 2011  http://class.ece.iastate.edu/cpre288  25
```

typedef

- `typedef` – a keyword used to assign alternative names to existing types
- By C coding convention, types defined with `typedef` should end with _t (examples: uint8_t, size_t)
- Chapter 6.7: C programming language

```
typedef char int8_t;
```

```
Fall 2011  http://class.ece.iastate.edu/cpre288  26
```

typedef examples

```
typedef char int8_t;

typedef struct RGB{
    int8_t red;
    int8_t green;
    int8_t blue;
} RGB_t;
```

```
LAB 2 OVERVIEW
```

```
typedef struct RGB{
    int8_t red;
    int8_t green;
    int8_t blue;
} RGB_t;
```

```
Fall 2011  http://class.ece.iastate.edu/cpre288  27
```

```
typedef examples
```

```
typedef char int8_t;

typedef struct RGB{
    int8_t red;
    int8_t green;
    int8_t blue;
} RGB_t;
```

```
LAB 2 OVERVIEW
```

```
typedef struct RGB{
    int8_t red;
    int8_t green;
    int8_t blue;
} RGB_t;
```

```
Fall 2011  http://class.ece.iastate.edu/cpre288  28
```

```
LAB 2 OVERVIEW
```

```
typedef struct RGB{
    int8_t red;
    int8_t green;
    int8_t blue;
} RGB_t;
```

```
Fall 2011  http://class.ece.iastate.edu/cpre288  29
```

```
LAB 2 OVERVIEW
```

```
typedef struct RGB{
    int8_t red;
    int8_t green;
    int8_t blue;
} RGB_t;
```

```
Fall 2011  http://class.ece.iastate.edu/cpre288  30
```

```
LAB 2 OVERVIEW
```
Open Interface

- Program is on the MCU (ATmega128 processor)
- Motors for movement are on the iRobot
- Communication occurs over a standard RS232 serial port using UART0
- This communication has been abstracted by using the open interface

Open Interface

- Open Interface makes it so you don’t have to “see” the serial communication
- You simply call functions that handle the serial part for you

```c
// Allocate a sensor struct
oi_t* oi_alloc();

// Initialize the serial communication
void oi_init(oi_t *self);

// Update the oi_t sensor struct
void oi_update(oi_t *self);

// Set velocity of each wheel in mm/s (value should be between -500 and +500)
void oi_set_wheels(int16_t right_wheel, int16_t left_wheel);
```

Open Interface

- Initializing the serial connection

```c
// Make sure the iRobot is powered on
oi_t* sensor_status = oi_alloc();    // allocate memory
oi_init(sensor_status);             // initialize
```

Open Interface

```c
typedef struct {
  // Boolean value for the right bumper
  uint8_t bumper_right;
  // Boolean value for the left bumper
  uint8_t bumper_left;
  // Boolean value for the right wheel
  uint8_t wheeldrop_right;
  // Boolean value for the left wheel
  uint8_t wheeldrop_left;
  // ... a lot more variables
} oi_t;
```

Move the Robot Forward

```c
#include “open_interface.h”
#include “util.h”

void main() {
  oi_t *robot = oi_alloc();
  oi_init(robot);

  ... // call a function to move robots

  free(robot);
}
```
#iRobot Open Interface and Movement

Lab 2, Part II. Robots moving in a square

Lab 2, Part III. Bump detection

What you will learn:
- How to program robot behavior using a set of API functions
- How API functions simplifies a programmer’s job

Common approaches when working with I/O devices

#Pointers

- What is a pointer?
Pointers: Mailbox Analogy

From Stoytchev's CprE 185 lecture notes

A letter fits comfortably in this box

A parcel does not. So, they give you a key ...

... the key opens a larger mailbox ...

... the parcel is stored there.

This is the pointer to the parcel.
Pointers

- A variable that stores the location (i.e. address) of another variable.

```c
int *pINT, XYZ;
char name[30], *pCHARARRAY;
pINT = &XYZ;
pCHARARRAY = name;
pCHARARRAY = &name[0];
XYZ = 10;
*pINT = 100;
*(pCHARARRAY+10) = 'A';
```

Pointers

- Pointers hold the address to another variable.
- You should understand these basic operations:

  **Operation**
  - Set the pointer to the address of a variable
  - Dereference the pointer
  - Set the value of the dereferenced object
  - Increment the pointer

  **Mailbox Analogy**
  - get the key for a certain mailbox
  - get the value of the parcel
  - set the value of the parcel
  - get the key for the next mailbox

- Pointers are declared using the * character

```c
int* ptr1; // pointer to type int
int *ptr2; // alternative declaration
char** ptr3; // pointer to type char
int** ptr4; // pointer to an int pointer
```

Pointers

- Setting the pointer to the address of a variable
  - & is the address operator
  - &myVariable is the address of myVariable
- Gets a mailbox address for a given parcel

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>OxFFFF</td>
<td></td>
</tr>
<tr>
<td>OxFFE</td>
<td></td>
</tr>
<tr>
<td>OxFFD</td>
<td></td>
</tr>
<tr>
<td>OxFFC</td>
<td></td>
</tr>
<tr>
<td>OxFFB</td>
<td></td>
</tr>
<tr>
<td>OxFFA</td>
<td></td>
</tr>
</tbody>
</table>
Pointers

- Setting the pointer to the address of a variable
  - & is the address operator
  - &myVariable is the address of myVariable

- Gets a mailbox address for a given parcel
  
<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFF</td>
<td>0x00</td>
</tr>
<tr>
<td>0xFFF</td>
<td>0x05</td>
</tr>
<tr>
<td>0xFFFD</td>
<td></td>
</tr>
<tr>
<td>0xFFFC</td>
<td></td>
</tr>
<tr>
<td>0xFFB</td>
<td></td>
</tr>
<tr>
<td>0xFFF</td>
<td></td>
</tr>
</tbody>
</table>

  \[\text{int } i = 5; \text{ int}* \text{ ip} = \&i;\]

  \[\text{int } \text{ x} = \*\text{ip}; \]
  \[\text{\text{\texttt{// x == i == 5}}};\]

  \[\text{http://class.ece.iastate.edu/cpre288}\]

Pointers

- Setting the pointer to the address of a variable
  - & is the address operator
  - &myVariable is the address of myVariable

- Gets a mailbox address for a given parcel
  
<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFF</td>
<td>0x00</td>
</tr>
<tr>
<td>0xFFF</td>
<td>0x05</td>
</tr>
<tr>
<td>0xFFFD</td>
<td></td>
</tr>
<tr>
<td>0xFFFC</td>
<td></td>
</tr>
<tr>
<td>0xFFB</td>
<td></td>
</tr>
<tr>
<td>0xFFF</td>
<td></td>
</tr>
</tbody>
</table>

  \[\text{int } i = 5; \text{ int}* \text{ ip} = \&i;\]

  \[\text{int } \text{ x} = \*\text{ip}; \]
  \[\text{\text{\texttt{// x == i == 5}}};\]

  \[\text{http://class.ece.iastate.edu/cpre288}\]

Pointers

- To dereference a pointer, use the * operator before the pointer's variable name

- Gets a parcel from a given mailbox address
  
<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFF</td>
<td>0x00</td>
</tr>
<tr>
<td>0xFFF</td>
<td>0x05</td>
</tr>
<tr>
<td>0xFFFD</td>
<td></td>
</tr>
<tr>
<td>0xFFFC</td>
<td></td>
</tr>
<tr>
<td>0xFFB</td>
<td></td>
</tr>
<tr>
<td>0xFFF</td>
<td></td>
</tr>
</tbody>
</table>

  \[\text{int } i = 5; \text{ int}* \text{ ip} = \&i; \text{ int } \text{ x} = \*\text{ip}; \text{ \texttt{// x == i == 5}}};\]

  \[\text{http://class.ece.iastate.edu/cpre288}\]

Pointers

- To dereference a pointer, use the * operator before the pointer's variable name

- Gets a parcel from a given mailbox address
  
<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFF</td>
<td>0x00</td>
</tr>
<tr>
<td>0xFFF</td>
<td>0x05</td>
</tr>
<tr>
<td>0xFFFD</td>
<td></td>
</tr>
<tr>
<td>0xFFFC</td>
<td></td>
</tr>
<tr>
<td>0xFFB</td>
<td></td>
</tr>
<tr>
<td>0xFFF</td>
<td></td>
</tr>
</tbody>
</table>

  \[\text{int } i = 5; \text{ int}* \text{ ip} = \&i; \text{ int } \text{ x} = \*\text{ip}; \text{ \texttt{// x == i == 5}}};\]

  \[\text{http://class.ece.iastate.edu/cpre288}\]
Pointers

• To set the value of i using the pointer, simply set the
dereferenced pointer
• Put a parcel in a certain mailbox
• In this case, *ip = 7 is equivalent to i = 7

```
int i = 5;
int* ip = &i;  // no dereference
*ip = 7;      // dereference and assign
```

• Think of the second statement as
  (int*) ip = &i;

```
0xFFFF 0x00
0xFFF0 0xFF
0xFFF1 0xFE
0xFFF2 0xFF
0xFFF3 0xFF
0xFFF4 0xFF
0xFFF5 0xFF
```

Pointers

• To set the value of i using the pointer, simply set the
dereferenced pointer
• Put a parcel in a certain mailbox
• In this case, *ip = 7 is equivalent to i = 7

```
int i = 5;
int* ip = &i;  // no dereference
*ip = 7;      // dereference and assign
```

• Think of the second statement as
  (int*) ip = &i;

```
0xFFFF 0x00
0xFFF0 0x07
0xFFF1 0xFF
0xFFF2 0xFE
0xFFF3 0xFF
0xFFF4 0xFF
```

Pointers

• Pointers can be reassigned to point to different objects
• Multiple pointers can point to the same object
• Pointers can point to memory space that exists outside
  your program or memory that doesn’t exist
  (causes an error)

```
int i = 5;
int* ip = &i;  // dereference and assign
*ip = 7;      // dereference and assign
```

• Think of the second statement as
  (int*) ip = &i;

```
0xFFFF 0x00
0xFFF0 0x07
0xFFF1 0xFF
0xFFF2 0xFE
0xFFF3 0xFF
0xFFF4 0xFF
```

```
0xFFFF 0x00
0xFFF0 0x07
0xFFF1 0xFF
0xFFF2 0xFE
0xFFF3 0xFF
0xFFF4 0xFF
```
Points

- Pointers can be reassigned to point to different objects
- Multiple pointers can point to the same object
- Pointers can point to memory space that exists outside your program or memory that doesn't exist (causes an error)

```
Address Value
0xFFFF 0x00
0xFFFE 0x07
0xFFFFD 0xFF
0xFFFC 0xFE
0xFFFFB 0x00
0xFFFFA 0x03
```

```
Address Value
0xFFFF 0x00
0xFFFE 0x07
0xFFFFD 0xFF
0xFFFC 0xFE
0xFFFFB 0x00
0xFFFFA 0x03
```

Pointers

- Pointers can be reassigned to point to different objects
- Multiple pointers can point to the same object
- Pointers can point to memory space that exists outside your program or memory that doesn't exist (causes an error)

```
Address Value
0xFFFF 0x00
0xFFFE 0x07
0xFFFFD 0xFF
0xFFFC 0xFE
0xFFFFB 0x00
0xFFFFA 0x03
```

```
Address Value
0xFFFF 0x00
0xFFFE 0x07
0xFFFFD 0xFF
0xFFFC 0xFE
0xFFFFB 0x00
0xFFFFA 0x03
```

Pointers

- Pointers can be reassigned to point to different objects
- Multiple pointers can point to the same object
- Pointers can point to memory space that exists outside your program or memory that doesn't exist (causes an error)

```
Address Value
0xFFFF 0x00
0xFFFE 0x07
0xFFFFD 0xFF
0xFFFC 0xFE
0xFFFFB 0x00
0xFFFFA 0x03
```

```
Address Value
0xFFFF 0x00
0xFFFE 0x07
0xFFFFD 0xFF
0xFFFC 0xFE
0xFFFFB 0x00
0xFFFFA 0x03
```

Pointers

- Incrementing and decrementing a pointer
  - Increments/decrements by the size of the type
- Example (on a byte addressed system)
  - int* increment by 2 (int's are 2 bytes on the ATmega 128)
  - char* increment by 1

```
Address Value
0xFFFF 0x00
0xFFFE 0x07
0xFFFFD 0xFF
0xFFFC 0xFE
0xFFFFB 0x00
0xFFFFA 0x03
```

```
Address Value
0xFFFF 0x00
0xFFFE 0x07
0xFFFFD 0xFF
0xFFFC 0xFE
0xFFFFB 0x00
0xFFFFA 0x03
```
Pointers

• Incrementing and decrementing a pointer
  — Increments/decrements by the size of the type
• Example (on a byte addressed system)
  — int* increment by 2 (int’s are 2 bytes on the ATmega 128)
  — char* increment by 1

```
int* ip = 0x1000;  // sizeof(int) == 2
char* cp = 0x1000; // sizeof(char) == 1
dp++;          
// ip == 0x1002 and cp = 0x1001
```

Pointers

• Incrementing and decrementing a pointer
  — Increments/decrements by the size of the type
• Example (on a byte addressed system)
  — int* increment by 2 (int’s are 2 bytes on the ATmega 128)
  — char* increment by 1

```
int* ip = 0x1000;  // sizeof(int) == 2
char* cp = 0x1000; // sizeof(char) == 1
dp++;          
// ip == 0x1002 and cp = 0x1001
```

Pointers

• Pointers are useful for passing parameters to a function
  by reference (instead of value)
  — Especially useful when the variables consume lots of memory
  — Java Objects use the same concept of pointers, as Objects are
  passed to functions by reference

```
void addThree(int *ptr) {
  *ptr = *ptr + 3;
}

void main() {
  int x = 5;
  addThree(&x);
  // x is now 8
}
```
Pass by Reference Example

void addThree(int *ptr) {
    *ptr = *ptr + 3;
}

void main() {
    int x = 5;
    addThree(&x);
    // x is now 8
}

---

Pass by Reference Example

void addThree(int *ptr) {
    *ptr = *ptr + 3;
}

void main() {
    int x = 5;
    addThree(&x);
    // x is now 8
}

---

Pointer Example

char s = 5;
char t = 8;
char *p1 = &s;
char **p2 = &p1;
*p1 = 9;
**p2 = 7;
*p2 = &t;
*p1 = 10;

- p1 points to s
- p2 points to p1
- Same as: s = 9;
- Same as: *p1 = 7; or s = 7;
- *p2 = &t; Same as: p1 now points to t
- *p1 = 10; Same as: t = 10;

---

Generic Processor Model

- Control signals
- n-bit processor: n is the width of register and data transfer, e.g. 32-bit

---

CPU, Memory, and Addresses

- Simplified Hardware picture showing how a CPU, Memory, and Addresses relate to each other
• Simplified Hardware picture showing how a CPU, Memory, and Addresses relate to each other

Assuming Stack order in Memory
char r = 0x10; // r is at 0xFFFF
char s = 0x15; // s is at 0xFFFE
char t = r; // t is at 0xFFFD

char t = r
char s =
char
Assuming Stack order in Memory
Memory, and Addresses relate to each other

Memory, and Addresses

char t = r
char s =
char
Assuming Stack order in Memory
Memory, and Addresses relate to each other

Memory, and Addresses

char t = r
char s =
char
Assuming Stack order in Memory
Memory, and Addresses relate to each other

Memory, and Addresses

char t = r
char s =
char
Assuming Stack order in Memory
Memory, and Addresses relate to each other
CPU, Memory, and Addresses

- Simplified Hardware picture showing how a CPU, Memory, and Addresses relate to each other

Assuming Stack order in Memory
char r = 0x10; // r is at 0xFFFF
char s = 0x15; // s is at 0xFFFE
char t = r; // t is at 0xFFFD

char *p1 = &s;
char *p2 = &t;
char **p3 = &p1;

Address     Value
r 0xFFFF 0x10
s 0xFFFE 0x15
t 0xFFFFD 0x10
p1 0xFFFC 0x15 0x10
p2 0xFFFFB 0xFFF9 0xFFF8 0xFFF7 0xFFF6 0xFFF5
p3 0x10

Pointer Example

char r = 10; // r is at 0xFFFF
char s = 15; // s is at 0xFFFE
char t = 13; // t is at 0xFFFD
char *p1 = &s;
char *p2 = &t;
char **p3 = &p1;

Address     Value
s 0xFFFE 0x15
p1 0xFFFFD 0x10
p2 0xFFFC 0x15 0x10
p3 0x10

CPU, Memory, and Addresses

- Simplified Hardware picture showing how a CPU, Memory, and Addresses relate to each other

Assuming Stack order in Memory
char r = 0x10; // r is at 0xFFFF
char s = 0x15; // s is at 0xFFFE
char t = r; // t is at 0xFFFD

char *p1 = &s;
char *p2 = &t;
char **p3 = &p1;

Address     Value
s 0xFFFE 0x15
p1 0xFFFFD 0x10
p2 0xFFFC 0x15 0x10
p3 0x10
char r = 10;
char s = 15;
char t = 13;

char *p1 = &s;
char *p2 = &t;
char **p3 = &p1;

*p1 = 20;
*p2 = 30;
**p3 = 40;
char *p1 = &s;
char *p2 = &t;
char **p3 = &p1;
*p3 = &p2;
*p3 = &r;

p3 = &p2;
*p3 = &r;

http://class.ece.iastate.edu/cpre288

// s = 20;
char s = 15;
char t = 13;
char *p1 = &s;
char *p2 = &t;
char **p3 = &p1;
p3 = &p2;
*p3 = &r;

http://class.ece.iastate.edu/cpre288
**Pointer Example**

```
char r = 10;
char s = 15;
char t = 13;
char *p1 = &s;
char *p2 = &t;
char **p3 = &p1;
p3 = &p2;
*p3 = &r;
*p1 = 20;
p2 = &r;
p1 = &t;
*p2 = 30;
*p3 = &t;
p3 = &p2;
*p3 = &r;
*p3 = 50;
*p3 = 40;
*p3 = 50;
*p3 = &p1;
*p3 = &r;
```
Exercise: Pointer

char msg[] = “Welcome to CprE 288”;
char *str;
Which of the following statements are good (valid and serve the purpose)?

a. str = msg[0];
b. str = msg;
c. str = &msg[10];
d. *str = msg;
e. *str = &msg[0];
f. *str = msg[10];

Assume the AVR platform, the variable addresses are assigned in stack order.

int x = 0x2050, y = 0x6633;
int* p1 = &x;
int* p2 = &y;
p2++;  (*p2 = *p1);

After executing the above code:

x   = ________  y   = ________
p1 = ________  p2 = ________

Assume the AVR platform, the variable addresses are assigned in stack order.

int x = 0x2050, y = 0x6633;
int* p1 = &x;
int* p2 = &y;
p2++;  (*p2 = *p1);

After executing the above code:

x   = ________  y   = ________
p1 = ________  p2 = ________
### Exercise: Pointer

Assume the AVR platform, the variable addresses are assigned in stack order.

```c
int x = 0x2050, y = 0x6633;
int* p1 = &x;
int* p2 = &y;
p2++; /*p2 = *p1;
```

After executing the above code:

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFF</td>
<td>0x20</td>
</tr>
<tr>
<td>0xFFFE</td>
<td>0x50</td>
</tr>
<tr>
<td>0xFFFD</td>
<td>0x66</td>
</tr>
<tr>
<td>0xFFFC</td>
<td>0x33</td>
</tr>
<tr>
<td>0xFFFFB</td>
<td>0xFE</td>
</tr>
<tr>
<td>0xFFFA</td>
<td>0xFFF</td>
</tr>
<tr>
<td>0xFFF9</td>
<td>0xFF</td>
</tr>
<tr>
<td>0xFFF8</td>
<td>0xFE</td>
</tr>
<tr>
<td>0xFFF7</td>
<td></td>
</tr>
<tr>
<td>0xFFF6</td>
<td></td>
</tr>
<tr>
<td>0xFFF5</td>
<td></td>
</tr>
</tbody>
</table>

x = ________
y = ________
p1 = ________
p2 = ________

### Exercise: Pointer, Array and Function

```c
int len;
char msg[] = "CPRE288 fun!";
Write a loop to calculate the length of msg and put it into len

a. Use pointer access
b. Use array access
```

### Reserved Words: Control Flow

- char
- double
- float
- int
- long
- short
- void
- enum
- struct
- union
- typedef
- break
- case
- continue
- default
- do
- else
- for
- goto
- if
- return
- switch
- while
- auto
- const
- extern
- register
- signed
- static
- unsigned
- volatile
- sizeof

### Control Flow in C

- Control Flow – Making the program behave in a particular manner depending on the input given to the program.
  - Why do we need Control Flow?
    - Not all program parts are executed all of the time, i.e., we want the program to intelligently choose what to do.
Control Flow in C

• REMEMBER! The evaluation for Boolean Control Flow is done on a TRUE / FALSE basis.
• TRUE / FALSE in the context of a computer is defined as
  – non-zero (TRUE)
  – zero (FALSE)

Examples:
-1, 5, 15, 225, 325.33 TRUE
0 FALSE

Control Flow in C: if, else if, else statement

Example
if (nVal > 10) {
  nVal += 5;
} else if (nVal > 5) {
  // If we reach this point, nVal must be <= 10
  nVal -= 3;
} else {
  // If we reach this point, nVal must be <= 10
  // and nVal must be <= 5
  nVal = 0;
}

Control Flow in C: If statement

• Must always have if statement; else if and else are optional

Follows a level hierarchy
• else if statements are only evaluated if all previous if and else if conditions have failed for the block
• else statements are only executed if all previous conditions have failed

Control Flow in C: comparison

Comparison (Relational Operators) – Numeric
>, >=
<, <=
== Equality
!= Not Equal

• Comparison expression gives a result of zero (FALSE) or non-zero (TRUE).
  – A TRUE result may not necessarily be a 1
  • Equality: Double equals sign ==
    – = Assigns a value
    – === Tests for equality, returns non-zero or zero

if (nVal == 5)  versus  if (nVal = 5)
The second expression always evaluates to TRUE. Why?

Control Flow in C: Boolean Logic

Comparison – Multiple Conditions
Tie together using Boolean (logical) operators
&& AND & bitwise
|| OR | bitwise
! NOT ~ bitwise

Examples:
if ( (nVal > 0) && (nArea < 10))
if( (nVal < 3) || (nVal > 50))
if ( ! (nVal <= 10) )

Control Flow in C: Boolean Logic

• WARNING!
  – Do not confuse bitwise AND, OR, and NOT operators with there Boolean counterparts
Control Flow in C: comparison

• Conditions are evaluated using lazy evaluation
  – Lazy evaluation – Once a condition is found that completes the condition, stop evaluating
  – OR any condition is found to be TRUE (1 OR'ed with anything = 1)
  – AND any condition is found to be FALSE (0 AND'ed with anything = 0)
• Why is lazy evaluation important?
  – Makes code run faster – skips unnecessary code. Once know condition will/will not evaluate, why evaluate other terms
• Can use lazy evaluation to guard against unwanted conditions
  – Checking for a NULL pointer before using the pointer
    if (str && *str != '\0')
• Why is lazy evaluation important?
  – Makes code run faster – skips unnecessary code. Once know condition will/will not evaluate, why evaluate other terms
  – OR any condition is found to be TRUE (1 OR'ed with anything = 1)
  – AND any condition is found to be FALSE (0 AND'ed with anything = 0)

Control Flow in C: Switch Statement

Switch statement

Ex: count zeros and ones
switch (n) {
  case 0:
    zero_counter++;
    break;
  case 1:
    one_counter++;
    break;
  default: // n is not equal to 0 or 1
    others_counter++;
}

Control Flow in C: For loop

// Syntax

for (initialization; conditional; loop) {
  /* loop body */
}
Control Flow in C: For loop

// Syntax
for (initialization; conditional; loop) {
    /* loop body */
}

// Best Practice
for (int i = 0; i < 10; i++) {
    // loop body
}

• The Initialization expression executes only once when first encountering the for loop.
• The Conditional expression executes at the beginning of each loop iteration; if false, control does not continue looping.
• The Loop expression execute at the end of each loop iteration.

Control Flow in C: For loop

// Equivalent loop with bad style
i = 0;
for (; i < 10;) {
    // loop body
    i++;
}

Control Flow in C: While loop

// Syntax
while (condition) {
    // loop body
}

// Example
int strlen(char *s) {
    int n = 0;  // string length
    while (s[n]) {
        n++;
    }
    return n;
}
Control Flow in C: do-while loop

// Syntax
do {
    // loop body
} while (condition);

Control Flow in C: Break statement

Break: Exit from the immediate for, do, while, or switch statement

int index = -1;
// Find the index of the "Lucky" element
for (i = 0; i < N; i++) {
    if (myNumbers[i] == 7) {
        index = i;
        break;
    }
}

• index contains the index of the element equal to 7, or index is -1 if no element equals 7

Control Flow in C: Continue statement

Continue statement: Start the next iteration of loop

for (i = 0; i < N; i++) {
    /* do pre-processing for all integers */
    ...
    if (X[i] < 0) {
        continue;
    }
    /* do post-processing for positives */
    ...
}

Control Flow in C: Goto statement

• Don’t use goto
  — Because Dijkstra says so
• Allows programmer to label code, then goto a spot in code using a goto label statement.

Do-while loop

int i = 0, sum = 0;
do {
    sum += X[i];
} while (i++ < N);

• Q: What’s the difference from the previous for loop?
  — A: The first iteration of the loop is always run, even if N is zero!