CprE 288 – Introduction to Embedded Systems

Instructors:
Dr. Phillip Jones
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)
• Control Flow (Chapter 3, C programming language)
  • for, if, else, switch, while, etc.
Announcements

• Homework 3 is due on Monday (Feb 8 – midnight) Black Board
• Lab 2 Demo in the first 20 minutes of Lab this week
• Lab 3 starts this week – Push Button, switch, stepper
FUNCTION CALLS
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!
Function Calls (short intro)

• All functions have
  – a return type (examples: char, void, int)
  – a name
  – a parameter list (or no parameters)
• Functions that have a return type (not void), should have a return statement

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

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

void main()
{
    int r = 5;
    r = add(3, 3);
    // r is now 6
}
Function Calls (short intro)

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

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

void main()
{
    int r = 5;
    r = add(3, 3);
    // r is now 6
}

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

```c
// my_functions.h
int add(int x, int y);
int sub(int x, int y);

#include “my_functions.h”;  // include a header file
void main()
{
    int r = 5;
    r = add(3, 3);
    // r is now 6
}

// my_functions.c
int add(int x, int y)
{
    return x + y;
}
```
ENUM, STRUCT, UNION, TYPEDEF
### 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
• 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

• http://en.wikipedia.org/wiki/Struct_(C_programming_language)
• Chapter 6.1 -6.5 (C programming Language)
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;
my_color_ptr->blue = 255;     // equivalent to previous line
```
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 : 1;
    unsigned int KBERROR : 1;
    unsigned int CAPLOCK : 1;
    unsigned int READY : 1;
} KBCR;
```
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: Merge multiple components

```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.
Union: Merge multiple components

```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.
This example the size is 4, since the largest component is 4 bytes.
• **Maintain multiple views:** byte or bit structure.

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

KBCR_U.KBCR_Aggregate = 0xc1;

KBCR_U.KBCR.READY = 1;
```
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?
Use of union inside of a struct

```c
struct {
    char *name;  2
    int flags;  2
    int utype;  2
    union {
        int ival;  2
        float fval; 4 //largest member of union u
        char *sval; 2
    } u; //largest member defines a union's size
} symtab;
```

Just sum the size of each struct member.

```
symtab size is: 2+2+2+4 = 10 bytes
```
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 examples

typedef char int8_t;

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

RGB_t my_color;
my_color.blue = 255;
typedef char int8_t;

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

// Array of RGB_t’s
RGB_t my_color[10]; // An array of 10 RGB_t
my_color[5].blue = 255; // set blue of 6th RGB_t
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
Open Interface

// 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
```
• **oi_t** sensor_status
  – it’s a struct for keeping the state of the iRobot
  – necessary since the status of sensors can only be current if serial communication is used
  – call **oi_update(sensor_status)**; to refresh the members of the struct

```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;
```
#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);
}
#include “open_interface.h”
#include “util.h”

int move_forward(oi_t *self, unsigned int distance_mm)
{
    oi_set_wheels(..., ...); // set the speed of both wheels
    int sum = 0;
    while (sum < distance_mm) {
        oi_update(self);
        sum += self->distance;
        // optional check for bump sensors
    }
    oi_set_wheels(..., ...); // stop the robot

    return sum;
}
Lab 2, Part II. Robots moving in a square
Lab 2, Part III. Bump detection
iRobot Open Interface and Movement

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

```c
int *pINT, XYZ;
pINT = &XYZ;
XYZ = 10;
*pINT = 100;
```

```c
char name[30], *pCHARARRAY;
pCHARARRAY = name;
pCHARARRAY = &name[0];
*(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>0xFFFF</td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>0xFFFE</td>
</tr>
<tr>
<td></td>
<td>0xFFFFD</td>
</tr>
<tr>
<td>ip</td>
<td>0xFFFFC</td>
</tr>
<tr>
<td></td>
<td>0xFFFB</td>
</tr>
<tr>
<td></td>
<td>0xFFFFA</td>
</tr>
</tbody>
</table>

int i = 5;
int* ip = &i;

[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>0x5</td>
</tr>
<tr>
<td>0xFFFFD</td>
<td></td>
</tr>
<tr>
<td>0xFFFEC</td>
<td></td>
</tr>
<tr>
<td>0xFFF FB</td>
<td></td>
</tr>
<tr>
<td>0xFFFFA</td>
<td></td>
</tr>
</tbody>
</table>

int i = 5;
int* ip = &i;
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>i</td>
<td>0xFFE</td>
</tr>
<tr>
<td></td>
<td>0xFFFFD</td>
</tr>
<tr>
<td>ip</td>
<td>0xFFF</td>
</tr>
<tr>
<td></td>
<td>0xFFFFB</td>
</tr>
<tr>
<td></td>
<td>0xFFFFA</td>
</tr>
</tbody>
</table>

int i = 5;
int* ip = &i;

[http://www.eskimo.com/~scs/cclass/notes/sx10a.html]
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>0xFFFFF</td>
<td></td>
</tr>
<tr>
<td>0xFFFFE</td>
<td></td>
</tr>
<tr>
<td>0xFFFFD</td>
<td></td>
</tr>
<tr>
<td>0xFFFC</td>
<td></td>
</tr>
<tr>
<td>0xFFFFB</td>
<td></td>
</tr>
<tr>
<td>0xFFFFA</td>
<td></td>
</tr>
</tbody>
</table>

```
int i = 5;
int* ip = &i;
int x = *ip;
// x == i == 5
```
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>i</td>
<td>0xFFFFE</td>
</tr>
<tr>
<td>ip</td>
<td>0xFFFFC</td>
</tr>
<tr>
<td></td>
<td>0xFFFFB</td>
</tr>
<tr>
<td>x</td>
<td>0xFFFFA</td>
</tr>
</tbody>
</table>

int i = 5;
int* ip = &i;
int x = *ip;
// x == i == 5
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>i</td>
<td>0xFFF8 0x05</td>
</tr>
<tr>
<td>ip</td>
<td>0xFFFFD 0xFF</td>
</tr>
<tr>
<td>x</td>
<td>0xFFFBA 0xFE</td>
</tr>
</tbody>
</table>

int i = 5;
int* ip = &i;
int x = *ip;
// x == i == 5
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>i</td>
<td>0xFFFFE 0x05</td>
</tr>
<tr>
<td></td>
<td>0xFFFDD 0xFF</td>
</tr>
<tr>
<td>ip</td>
<td>0xFFFDC 0xFE</td>
</tr>
<tr>
<td></td>
<td>0xFFFFB 0x00</td>
</tr>
<tr>
<td>x</td>
<td>0xFFFFA 0x05</td>
</tr>
</tbody>
</table>

int i = 5;
int* ip = &i;
int x = *ip;
// x == i == 5
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

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFF</td>
<td></td>
</tr>
<tr>
<td>i</td>
<td></td>
</tr>
<tr>
<td>0xFFFE</td>
<td></td>
</tr>
<tr>
<td>0xFFFFD</td>
<td></td>
</tr>
<tr>
<td>ip</td>
<td></td>
</tr>
<tr>
<td>0xFFFFC</td>
<td></td>
</tr>
<tr>
<td>0xFFFB</td>
<td></td>
</tr>
<tr>
<td>0xFFFFA</td>
<td></td>
</tr>
</tbody>
</table>

```c
int i = 5;
int* ip = &i;
*ip = 7;
```
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;
*ip = 7;
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFF</td>
<td>0x00</td>
</tr>
<tr>
<td>0xFFE</td>
<td>0x05</td>
</tr>
<tr>
<td>0xFFFFD</td>
<td></td>
</tr>
<tr>
<td>0xFFFCE</td>
<td></td>
</tr>
<tr>
<td>0xFFFBD</td>
<td></td>
</tr>
<tr>
<td>0xFFFBA</td>
<td></td>
</tr>
<tr>
<td>0xFFFFA</td>
<td></td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFFF</td>
<td>0x00</td>
</tr>
<tr>
<td>0xFFFFE</td>
<td>0x05</td>
</tr>
<tr>
<td>0xFFFFD</td>
<td>0xFF</td>
</tr>
<tr>
<td>0xFFFFC</td>
<td>0xFE</td>
</tr>
<tr>
<td>0xFFFFB</td>
<td></td>
</tr>
<tr>
<td>0xFFFA</td>
<td></td>
</tr>
</tbody>
</table>

```c
int i = 5;
int* ip = &i;
*ip = 7;
```
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;
*ip = 7;
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFF</td>
<td>0x00</td>
</tr>
<tr>
<td>i</td>
<td></td>
</tr>
<tr>
<td>0xFFFE</td>
<td>0x07</td>
</tr>
<tr>
<td>0xFFFFD</td>
<td>0xFF</td>
</tr>
<tr>
<td>ip</td>
<td></td>
</tr>
<tr>
<td>0xFFFFC</td>
<td>0xFE</td>
</tr>
<tr>
<td>0xFFFB</td>
<td></td>
</tr>
<tr>
<td>0xFFFFFA</td>
<td></td>
</tr>
</tbody>
</table>

http://class.ece.iastate.edu/cpre288
• **WARNING!** A * operator is used for both dereferencing and for declaring a pointer.

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

• Think of the second statement as

```c
(int*) ip = &i;
```
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;
*ip = 7;
int j = 3;
ip = &j;
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFF</td>
<td></td>
</tr>
<tr>
<td>0xFFFFE</td>
<td></td>
</tr>
<tr>
<td>0xFFFFD</td>
<td></td>
</tr>
<tr>
<td>0xFFFFC</td>
<td></td>
</tr>
<tr>
<td>0xFFFFB</td>
<td></td>
</tr>
<tr>
<td>0xFFFFA</td>
<td></td>
</tr>
</tbody>
</table>

• 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)

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFF</td>
<td>0x00</td>
</tr>
<tr>
<td>0xFFFD</td>
<td></td>
</tr>
<tr>
<td>0xFFFC</td>
<td></td>
</tr>
<tr>
<td>0xFFFB</td>
<td></td>
</tr>
<tr>
<td>0xFFFFA</td>
<td></td>
</tr>
</tbody>
</table>

```c
int i = 5;
int* ip = &i;
*ip = 7;
int j = 3;
ip = &j;
```
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)

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFFF</td>
<td>0x00</td>
</tr>
<tr>
<td>0xFFFE</td>
<td>0x05</td>
</tr>
<tr>
<td>0xFFFFD</td>
<td>0xFF</td>
</tr>
<tr>
<td>0xFFFFC</td>
<td>0xFE</td>
</tr>
<tr>
<td>0xFFFFB</td>
<td></td>
</tr>
<tr>
<td>0xFFFA</td>
<td></td>
</tr>
</tbody>
</table>

```c
int i = 5;
int* ip = &i;
*ip = 7;
int j = 3;
ip = &j;
```
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)

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFF</td>
<td>0x00</td>
</tr>
<tr>
<td>0xFFFE</td>
<td>0x07</td>
</tr>
<tr>
<td>0xFFFFD</td>
<td>0xFF</td>
</tr>
<tr>
<td>0xFFFFC</td>
<td>0xFE</td>
</tr>
<tr>
<td>0xFFFFB</td>
<td></td>
</tr>
<tr>
<td>0xFFFFA</td>
<td></td>
</tr>
</tbody>
</table>

```
int i = 5;
int* ip = &i;
*ip = 7;
int j = 3;
ip = &j;
```
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;
/ip = 7;
int j = 3;
ip = &j;
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFF</td>
<td>0x00</td>
</tr>
<tr>
<td>0xFFFFE</td>
<td>0x07</td>
</tr>
<tr>
<td>0xFFFFD</td>
<td>0xFF</td>
</tr>
<tr>
<td>0xFFFFC</td>
<td>0xFE</td>
</tr>
<tr>
<td>0xFFFFB</td>
<td>0x00</td>
</tr>
<tr>
<td>0xFFFA</td>
<td>0x03</td>
</tr>
</tbody>
</table>
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;
*ip = 7;
int j = 3;
ip = &j;
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFF</td>
<td>0x00</td>
</tr>
<tr>
<td>i</td>
<td></td>
</tr>
<tr>
<td>0xFFFEE</td>
<td>0x07</td>
</tr>
<tr>
<td>0xFFFFD</td>
<td>0xFF</td>
</tr>
<tr>
<td>ip</td>
<td></td>
</tr>
<tr>
<td>0xFFFFC</td>
<td>0xFA</td>
</tr>
<tr>
<td>0xFFFFB</td>
<td>0x00</td>
</tr>
<tr>
<td>j</td>
<td></td>
</tr>
<tr>
<td>0xFFFFA</td>
<td>0x03</td>
</tr>
</tbody>
</table>
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

```c
int* ip = 0x1000;  // sizeof(int) == 2
char* cp = 0x1000;  // sizeof(char) == 1
ip++;  
cp++;  
// 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

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFF</td>
<td>0x10</td>
</tr>
<tr>
<td>0xFFFE</td>
<td>0x00</td>
</tr>
<tr>
<td>0xFFFFD</td>
<td></td>
</tr>
<tr>
<td>0xFFFC</td>
<td></td>
</tr>
<tr>
<td>0xFFFFB</td>
<td></td>
</tr>
<tr>
<td>0xFFFFA</td>
<td></td>
</tr>
</tbody>
</table>

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

• 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

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFF</td>
<td>0x10</td>
</tr>
<tr>
<td>0xFFFE</td>
<td>0x00</td>
</tr>
<tr>
<td>0xFFFFD</td>
<td>0x10</td>
</tr>
<tr>
<td>0xFFFC</td>
<td>0x00</td>
</tr>
<tr>
<td>0xFFFFB</td>
<td></td>
</tr>
<tr>
<td>0xFFFFA</td>
<td></td>
</tr>
</tbody>
</table>
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

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFF</td>
<td>0x10</td>
</tr>
<tr>
<td>0xFFFE</td>
<td>0x02</td>
</tr>
<tr>
<td>0xFFFFD</td>
<td>0x10</td>
</tr>
<tr>
<td>0xFFFFC</td>
<td>0x00</td>
</tr>
<tr>
<td>0xFFFFB</td>
<td></td>
</tr>
<tr>
<td>0xFFFFA</td>
<td></td>
</tr>
</tbody>
</table>
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
ip++;              
cp++;              
  // ip == 0x1002 and cp = 0x1001
```

```
<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFF</td>
<td>0x10</td>
</tr>
<tr>
<td>0xFFFE</td>
<td>0x02</td>
</tr>
<tr>
<td>0xFFFFD</td>
<td>0x10</td>
</tr>
<tr>
<td>0xFFFC</td>
<td>0x01</td>
</tr>
<tr>
<td>0xFFFFB</td>
<td></td>
</tr>
<tr>
<td>0xFFFFA</td>
<td></td>
</tr>
</tbody>
</table>
```
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
Pass by Reference Example

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

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFFF</td>
<td></td>
</tr>
<tr>
<td>0xFFFFE</td>
<td></td>
</tr>
<tr>
<td>0xFFFD</td>
<td></td>
</tr>
<tr>
<td>0xFFFFC</td>
<td></td>
</tr>
<tr>
<td>0xFFFB</td>
<td></td>
</tr>
<tr>
<td>0xFFF9</td>
<td></td>
</tr>
<tr>
<td>0xFFF8</td>
<td></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>
void addThree(int *ptr) {
    *ptr = *ptr + 3;
}

void main() {
    int x = 5;
    addThree(&x);
    // x is now 8
}
void addThree(int *ptr) {
    *ptr = *ptr + 3;
}

void main() {
    int x = 5;
    addThree(&x);
    // x is now 8
}
void addThree(int *ptr) {
    *ptr = *ptr + 3;
}

void main() {
    int x = 5;
    addThree(&x);
    // x is now 8
}
char s = 5;
char t = 8;
char *p1 = &s;
char **p2 = &p1;

• p1 points to s
• p2 points to p1

*p1 = 9;
**p2 = 7;
*p2 = &t;
*p1 = 10;

• Same as: s = 9;
• Same as: *p1 = 7; or s = 7;
• Same as: p1 = &t; (p1 now points to t)
• 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
• 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

http://class.ece.iastate.edu/cpre288
• Simplified Hardware picture showing how a CPU, Memory, and Addresses relate to each other

Assuming Stack order in Memory

```c
char r = 0x10; // r is at 0xFFFF
cchar s = 0x15; // s is at 0xFFFE
cchar t = r; // t is at 0xFFFD
```
• Simplified Hardware picture showing how a CPU, Memory, and Addresses relate to each other

Assuming Stack order in Memory

```c
char r = 0x10; // r is at 0xFFFF
char s = 0x15; // s is at 0xFFFE
char t = r; // t is at 0xFFFD
```
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
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

0xFFFF = Write Address
0x15 = Write Data
CPU, Memory, and Addresses

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

Assuming Stack order in Memory

```c
char r = 0x10; // r is at 0xFFFF
char s = 0x15; // s is at 0xFFFE
char t = r; // t is at 0xFFFF
```
• 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
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 0xFFFFD
• 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

0xFFFF = 0x10 = 0xFFFD
0xFFFE = 0x15
0xFFFD = Write Address
0x10 = Write Data
0xFFFF = Read Address = 0xFFFF
0x10 = Read Data
• 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
### Pointer Example

```c
char r = 10;
char s = 15;
char t = 13;
char *p1 = &s;
char *p2 = &t;
char **p3 = &p1;

*p1 = 20;
*p2 = 30;
**p3 = 40;
*p3 = &t;
**p3 = 50;
p3 = &p2;
*p3 = &r;
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>0xFFFFF</td>
</tr>
<tr>
<td>s</td>
<td>0xFFFFE</td>
</tr>
<tr>
<td>t</td>
<td>0xFFFFD</td>
</tr>
<tr>
<td>p1</td>
<td>0xFFFFC</td>
</tr>
<tr>
<td>p2</td>
<td>0xFFFFB</td>
</tr>
<tr>
<td>p3</td>
<td>0xFFFA</td>
</tr>
<tr>
<td>p2</td>
<td>0xFFF9</td>
</tr>
<tr>
<td>p2</td>
<td>0xFFF8</td>
</tr>
<tr>
<td>p3</td>
<td>0xFFF7</td>
</tr>
<tr>
<td>p3</td>
<td>0xFFF6</td>
</tr>
<tr>
<td>p3</td>
<td>0xFFF5</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>r 0xFFFF</td>
<td>0x0A</td>
</tr>
<tr>
<td>s 0xFFFFE</td>
<td></td>
</tr>
<tr>
<td>t 0xFFFFD</td>
<td></td>
</tr>
<tr>
<td>p1 0xFFFC</td>
<td></td>
</tr>
<tr>
<td>p2 0xFFF9</td>
<td></td>
</tr>
<tr>
<td>p3 0xFFF7</td>
<td></td>
</tr>
</tbody>
</table>

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

```c
char r = 10;
char s = 15;
char t = 13;
char *p1 = &s;
char *p2 = &t;
char **p3 = &p1;

*p1 = 20;
p2 = 30;
**p3 = 40;
p3 = &t;
**p3 = 50;
p3 = &p2;
*p3 = &r;
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFF</td>
<td>0xA</td>
</tr>
<tr>
<td>0xFFF8</td>
<td>0xF</td>
</tr>
<tr>
<td>0xFFF7</td>
<td>0xF</td>
</tr>
<tr>
<td>0xFFF6</td>
<td>0xF</td>
</tr>
<tr>
<td>0xFFF5</td>
<td>0xF</td>
</tr>
<tr>
<td>0xFFF4</td>
<td>0xF</td>
</tr>
<tr>
<td>0xFFF3</td>
<td>0xF</td>
</tr>
<tr>
<td>0xFFF2</td>
<td>0xF</td>
</tr>
<tr>
<td>0xFFF1</td>
<td>0xF</td>
</tr>
<tr>
<td>0xFFF0</td>
<td>0xF</td>
</tr>
<tr>
<td>0xFFF9</td>
<td>0xF</td>
</tr>
<tr>
<td>0xFFF8</td>
<td>0xF</td>
</tr>
<tr>
<td>0xFFF7</td>
<td>0xF</td>
</tr>
<tr>
<td>0xFFF6</td>
<td>0xF</td>
</tr>
<tr>
<td>0xFFF5</td>
<td>0xF</td>
</tr>
</tbody>
</table>
```
char r = 10;
char s = 15;
char t = 13;
char *p1 = &s;
char *p2 = &t;
char **p3 = &p1;
*p1 = 20;
*p2 = 30;
**p3 = 40;
*p3 = &t;
**p3 = 50;
p3 = &p2;
*p3 = &r;

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFF</td>
<td>0x0A</td>
</tr>
<tr>
<td>0xFFFE</td>
<td>0x0F</td>
</tr>
<tr>
<td>0xFFFD</td>
<td>0x0D</td>
</tr>
<tr>
<td>0xFFFC</td>
<td></td>
</tr>
<tr>
<td>0xFFFB</td>
<td></td>
</tr>
<tr>
<td>0xFFFFA</td>
<td></td>
</tr>
<tr>
<td>0xFFF9</td>
<td></td>
</tr>
<tr>
<td>0xFFF8</td>
<td></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>
### Pointer Example

```c
char r = 10;
char s = 15;
char t = 13;
char *p1 = &s;
char *p2 = &t;
char **p3 = &p1;

*p1 = 20;
*p2 = 30;
**p3 = 40;
*p3 = &t;
**p3 = 50;
p3 = &p2;
*p3 = &r;
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>0xFFFF</td>
</tr>
<tr>
<td>s</td>
<td>0xFFFE</td>
</tr>
<tr>
<td>t</td>
<td>0xFFFFD</td>
</tr>
<tr>
<td>**p3</td>
<td>0xFFFFC</td>
</tr>
<tr>
<td>p1</td>
<td>0xFFFFB</td>
</tr>
<tr>
<td>**p3</td>
<td>0xFFFFA</td>
</tr>
<tr>
<td>p2</td>
<td>0xFFF9</td>
</tr>
<tr>
<td>**p3</td>
<td>0xFFF8</td>
</tr>
<tr>
<td>p3</td>
<td>0xFFF7</td>
</tr>
<tr>
<td>**p3</td>
<td>0xFFF6</td>
</tr>
<tr>
<td>**p3</td>
<td>0xFFF5</td>
</tr>
</tbody>
</table>
```

---

[Link to course material](http://class.ece.iastate.edu/cpre288)
char r = 10;
char s = 15;
char t = 13;
char *p1 = &s;
char *p2 = &t;
char **p3 = &p1;

*p1 = 20;
*p2 = 30;
**p3 = 40;
*p3 = &t;
**p3 = 50;
p3 = &p2;
*p3 = &r;

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFF</td>
<td>0x0A</td>
</tr>
<tr>
<td>0xFFFE</td>
<td>0x0F</td>
</tr>
<tr>
<td>0xFFFFD</td>
<td>0x0D</td>
</tr>
<tr>
<td>0xFFFFC</td>
<td>0xFF</td>
</tr>
<tr>
<td>0xFFFB</td>
<td>0xFE</td>
</tr>
<tr>
<td>0xFFF9</td>
<td>0xFD</td>
</tr>
<tr>
<td>0xFFF8</td>
<td></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>
char r = 10;
char s = 15;
char t = 13;
char *p1 = &s;
char *p2 = &t;
char **p3 = &p1;

*p1 = 20;
*p2 = 30;
**p3 = 40;
*p3 = &t;
**p3 = 50;
p3 = &p2;
*p3 = &r;

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFF</td>
<td>0xA0</td>
</tr>
<tr>
<td>0xFFE</td>
<td>0x0F</td>
</tr>
<tr>
<td>0xFFD</td>
<td>0xD0</td>
</tr>
<tr>
<td>0xFFC</td>
<td>0xFF</td>
</tr>
<tr>
<td>0xFB</td>
<td>0xEE</td>
</tr>
<tr>
<td>0xF9</td>
<td>0xFD</td>
</tr>
<tr>
<td>0xF8</td>
<td>0xFF</td>
</tr>
<tr>
<td>0xF7</td>
<td>0xFB</td>
</tr>
<tr>
<td>0xF6</td>
<td></td>
</tr>
<tr>
<td>0xF5</td>
<td></td>
</tr>
</tbody>
</table>
char r = 10;  // s = 20;
char s = 15;
char t = 13;
char *p1 = &s;
char *p2 = &t;
char **p3 = &p1;

*p1 = 20;
*p2 = 30;
**p3 = 40;
*p3 = &t;
**p3 = 50;
p3 = &p2;
*p3 = &r;

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFF</td>
<td>0xA</td>
</tr>
<tr>
<td>0xFFE</td>
<td>0x14</td>
</tr>
<tr>
<td>0xFFD</td>
<td>0xD</td>
</tr>
<tr>
<td>0xFFC</td>
<td>0xF</td>
</tr>
<tr>
<td>0xFFB</td>
<td>0xE</td>
</tr>
<tr>
<td>0xFFF</td>
<td>0xF</td>
</tr>
<tr>
<td>0xFFF9</td>
<td>0xFD</td>
</tr>
<tr>
<td>0xFFF8</td>
<td>0xFF</td>
</tr>
<tr>
<td>0xFFF7</td>
<td>0xFB</td>
</tr>
<tr>
<td>0xFFF6</td>
<td></td>
</tr>
<tr>
<td>0xFFF5</td>
<td></td>
</tr>
</tbody>
</table>
```c
char r = 10;   *p1 = 20;   // s = 20;
char s = 15;   *p2 = 30;   // t = 30;
char t = 13;   **p3 = 40;
char *p1 = &s; *p3 = &t;
char *p2 = &t; **p3 = 50;
char **p3 = &p1;
    p3 = &p2;
    *p3 = &r;
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFE</td>
<td>0x14</td>
</tr>
<tr>
<td>0xFFFFD</td>
<td>0x1E</td>
</tr>
<tr>
<td>0xFFFDC</td>
<td>0xFF</td>
</tr>
<tr>
<td>0xFFFBB</td>
<td>0xFE</td>
</tr>
<tr>
<td>0xFFFAC</td>
<td>0xFF</td>
</tr>
<tr>
<td>0xFFF9B</td>
<td>0xFD</td>
</tr>
<tr>
<td>0xFFF8A</td>
<td>0xFF</td>
</tr>
<tr>
<td>0xFFF7B</td>
<td>0xFB</td>
</tr>
<tr>
<td>0xFFF6A</td>
<td></td>
</tr>
<tr>
<td>0xFFF5A</td>
<td></td>
</tr>
</tbody>
</table>

http://class.ece.iastate.edu/cpre288
char r = 10;  // s = 20;
char s = 15;  // t = 30;
char t = 13;
char *p1 = &s;
char *p2 = &t;
char **p3 = &p1;
char **p3 = &p1;
*p1 = 20;
*p2 = 30;
**p3 = 40;
*p3 = &t;
**p3 = 50;
p3 = &p2;
*p3 = &r;

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFF</td>
<td>0x0A</td>
</tr>
<tr>
<td>0xFFFE</td>
<td>0x28</td>
</tr>
<tr>
<td>0xFFFFD</td>
<td>0x1E</td>
</tr>
<tr>
<td>0xFFFC</td>
<td>0xFF</td>
</tr>
<tr>
<td>0xFFFFB</td>
<td>0xFE</td>
</tr>
<tr>
<td>0xFFFFA</td>
<td>0xFF</td>
</tr>
<tr>
<td>0xFFF9</td>
<td>0xFD</td>
</tr>
<tr>
<td>0xFFF8</td>
<td>0xFF</td>
</tr>
<tr>
<td>0xFFF7</td>
<td>0xFB</td>
</tr>
<tr>
<td>0xFFF6</td>
<td></td>
</tr>
<tr>
<td>0xFFF5</td>
<td></td>
</tr>
</tbody>
</table>
### Pointer Example

```c
char r = 10;
char s = 15;
char t = 13;
char *p1 = &s;
char *p2 = &t;
char **p3 = &p1;

*p1 = 20;  // s = 20;
*p2 = 30;  // t = 30;
**p3 = 40; // s = 40;
*p3 = &t;  // p1 = &t;
**p3 = 50;
p3 = &p2;
*p3 = &r;
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFF</td>
<td>0xA</td>
</tr>
<tr>
<td>0xFFFE</td>
<td>0x28</td>
</tr>
<tr>
<td>0xFFFFD</td>
<td>0x1E</td>
</tr>
<tr>
<td>0xFFFCE</td>
<td>0xFF</td>
</tr>
<tr>
<td>0xFFFBB</td>
<td>0xFD</td>
</tr>
<tr>
<td>0xFFF9</td>
<td>0xFD</td>
</tr>
<tr>
<td>0xFFF8</td>
<td>0xFF</td>
</tr>
<tr>
<td>0xFFF7</td>
<td>0xFB</td>
</tr>
<tr>
<td>0xFFF6</td>
<td></td>
</tr>
<tr>
<td>0xFFF5</td>
<td></td>
</tr>
</tbody>
</table>
### Pointer Example

```c
char r = 10;
char s = 15;
char t = 13;
char *p1 = &s;
char *p2 = &t;
char **p3 = &p1;
p3 = &p2;
*p3 = &r;

*p1 = 20;  // s = 20;
*p2 = 30;  // t = 30;
**p3 = 40; // s = 40;
**p3 = 50; // t = 50;

// s = 20;
// t = 30;
// s = 40;
// p1 = &t;
// t = 50;
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFF</td>
<td>0x0A</td>
</tr>
<tr>
<td>0xFFFFE</td>
<td>0x28</td>
</tr>
<tr>
<td>0xFFFFD</td>
<td>0x32</td>
</tr>
<tr>
<td>0xFFFEC</td>
<td>0xFF</td>
</tr>
<tr>
<td>0xFFFDB</td>
<td>0xFD</td>
</tr>
<tr>
<td>0xFFFDA</td>
<td>0xFF</td>
</tr>
<tr>
<td>0xFFF9</td>
<td>0xFD</td>
</tr>
<tr>
<td>0xFFF8</td>
<td>0xFF</td>
</tr>
<tr>
<td>0xFFF7</td>
<td>0xFB</td>
</tr>
<tr>
<td>0xFFF6</td>
<td></td>
</tr>
<tr>
<td>0xFFF5</td>
<td></td>
</tr>
</tbody>
</table>

http://class.ece.iastate.edu/cpre288
char r = 10;
char s = 15;
char t = 13;
char *p1 = &s;
char *p2 = &t;
char **p3 = &p1;
*p1 = 20;  // s = 20;
*p2 = 30;  // t = 30;
**p3 = 40; // s = 40;
*p3 = &t;  // p1 = &t;
**p3 = 50; // t = 50;
p3 = &p2;
*p3 = &r;

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFF</td>
<td>0xA</td>
</tr>
<tr>
<td>0xFFF9</td>
<td>0xF9</td>
</tr>
<tr>
<td>0xFFF8</td>
<td>0xFF</td>
</tr>
<tr>
<td>0xFFF7</td>
<td>0xF9</td>
</tr>
<tr>
<td>0xFFF6</td>
<td></td>
</tr>
<tr>
<td>0xFFF5</td>
<td></td>
</tr>
<tr>
<td>0xFFF4</td>
<td></td>
</tr>
<tr>
<td>0xFFF3</td>
<td></td>
</tr>
<tr>
<td>0xFFF2</td>
<td></td>
</tr>
<tr>
<td>0xFFF1</td>
<td></td>
</tr>
<tr>
<td>0xFFF0</td>
<td></td>
</tr>
<tr>
<td>0xFFF6</td>
<td>0xFD</td>
</tr>
<tr>
<td>0xFFFB</td>
<td>0xFD</td>
</tr>
<tr>
<td>0xFFFC</td>
<td>0xFF</td>
</tr>
<tr>
<td>0xFFFD</td>
<td>0x32</td>
</tr>
<tr>
<td>0xFFFE</td>
<td>0x28</td>
</tr>
<tr>
<td>0xFFFF</td>
<td>0xA</td>
</tr>
</tbody>
</table>
**Pointer Example**

```c
char r = 10;
char s = 15;
char t = 13;
char *p1 = &s;
char *p2 = &t;
char **p3 = &p1;

*p1 = 20;  // s = 20;
*p2 = 30;  // t = 30;
**p3 = 40; // s = 40;
*p3 = &t;  // p1 = &t;
**p3 = 50; // t = 50;
p3 = &p2;
*p3 = &r;  // p2 = &r;
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFFF</td>
<td>0xA</td>
</tr>
<tr>
<td>0xFFFE</td>
<td>0x28</td>
</tr>
<tr>
<td>0xFFFFD</td>
<td>0x32</td>
</tr>
<tr>
<td>0xFFFFC</td>
<td>0xFF</td>
</tr>
<tr>
<td>0xFFFB</td>
<td>0xFD</td>
</tr>
<tr>
<td>0xFFFA</td>
<td>0xFF</td>
</tr>
<tr>
<td>0xFFF9</td>
<td>0xFF</td>
</tr>
<tr>
<td>0xFFF8</td>
<td>0xFF</td>
</tr>
<tr>
<td>0xFFF7</td>
<td>0xF9</td>
</tr>
<tr>
<td>0xFFF6</td>
<td></td>
</tr>
<tr>
<td>0xFFF5</td>
<td></td>
</tr>
</tbody>
</table>
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];
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></td>
</tr>
<tr>
<td>0xFFFE</td>
<td></td>
</tr>
<tr>
<td>0xFFFFD</td>
<td></td>
</tr>
<tr>
<td>0xFFFC</td>
<td></td>
</tr>
<tr>
<td>0xFFFB</td>
<td></td>
</tr>
<tr>
<td>0xFFFFA</td>
<td></td>
</tr>
<tr>
<td>0xFFFF9</td>
<td></td>
</tr>
<tr>
<td>0xFFFF8</td>
<td></td>
</tr>
<tr>
<td>0xFFFF7</td>
<td></td>
</tr>
<tr>
<td>0xFFFF6</td>
<td></td>
</tr>
<tr>
<td>0xFFFF5</td>
<td></td>
</tr>
</tbody>
</table>

x = \_

y = \_

p1 = \_

p2 = \_

May 18, 2011

http://class.ece.iastate.edu/cpre288
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>0xFFFFD</td>
<td></td>
</tr>
<tr>
<td>0xFFFEC</td>
<td></td>
</tr>
<tr>
<td>0xFFFBD</td>
<td></td>
</tr>
<tr>
<td>0xFFFAC</td>
<td></td>
</tr>
<tr>
<td>0xFFF9A</td>
<td></td>
</tr>
<tr>
<td>0xFFF8A</td>
<td></td>
</tr>
<tr>
<td>0xFFF7A</td>
<td></td>
</tr>
<tr>
<td>0xFFF6A</td>
<td></td>
</tr>
</tbody>
</table>

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>0xFFFFF</td>
<td>0x20</td>
</tr>
<tr>
<td>0xFFFE</td>
<td>0x50</td>
</tr>
<tr>
<td>0xFFFFD</td>
<td>0x66</td>
</tr>
<tr>
<td>0xFFFFC</td>
<td>0x33</td>
</tr>
<tr>
<td>0xFFFB</td>
<td></td>
</tr>
<tr>
<td>0xFFFFA</td>
<td></td>
</tr>
<tr>
<td>0xFFFF9</td>
<td></td>
</tr>
<tr>
<td>0xFFFF8</td>
<td></td>
</tr>
<tr>
<td>0xFFFF7</td>
<td></td>
</tr>
<tr>
<td>0xFFFF6</td>
<td></td>
</tr>
<tr>
<td>0xFFFF5</td>
<td></td>
</tr>
</tbody>
</table>

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>0xFFFFD</td>
<td>0x66</td>
</tr>
<tr>
<td>0xFFFC</td>
<td>0x33</td>
</tr>
<tr>
<td>0xFFFFB</td>
<td>0xFF</td>
</tr>
<tr>
<td>0xFFFFA</td>
<td>0xFE</td>
</tr>
<tr>
<td>0xFFF9</td>
<td></td>
</tr>
<tr>
<td>0xFFF8</td>
<td></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 = ________
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;
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFFF</td>
<td>0x20</td>
</tr>
<tr>
<td>0xFFFE</td>
<td>0x50</td>
</tr>
<tr>
<td>0xFFFFD</td>
<td>0x66</td>
</tr>
<tr>
<td>0xFFFC</td>
<td>0x33</td>
</tr>
<tr>
<td>0xFFFB</td>
<td>0xFFF</td>
</tr>
<tr>
<td>0xFFF9</td>
<td>0xFE</td>
</tr>
<tr>
<td>0xFFF8</td>
<td>0xFC</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>

After executing the above code:

- `x` = 
- `y` = 
- `p1` = 
- `p2` = 

May 18, 2011
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++;  // Corrected from p2 += 1
*p2 = *p1;
```

After executing the above code:

- `x` = ________
- `y` = ________
- `p1` = ________
- `p2` = ________

---

### Memory Address Table

<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>0xFFFFD</td>
<td>0x66</td>
</tr>
<tr>
<td>0xFFFC</td>
<td>0x33</td>
</tr>
<tr>
<td>0xFFFB</td>
<td>0xFF</td>
</tr>
<tr>
<td>0xFFFFA</td>
<td>0xFE</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>

May 18, 2011
Exercise: Pointer

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** = ________

<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>0xFFFFD</td>
<td>0x66</td>
</tr>
<tr>
<td>0xFFFCC</td>
<td>0x33</td>
</tr>
<tr>
<td>0xFFFBD</td>
<td>0xFFF</td>
</tr>
<tr>
<td>0xFFFBA</td>
<td>0xFE</td>
</tr>
<tr>
<td>0xFFF9</td>
<td>0xFFF</td>
</tr>
<tr>
<td>0xFFF7</td>
<td>0xFE</td>
</tr>
<tr>
<td>0xFFF6</td>
<td></td>
</tr>
<tr>
<td>0xFFF5</td>
<td></td>
</tr>
</tbody>
</table>
int len;
char msg[] = “CPRE288 fun!”;

Write a loop to calculate the length of \textit{msg} and put it into \textit{len}

a. Use pointer access
b. Use array access
CONTROL FLOW IN C
### 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
Example

```c
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

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

http://class.ece.iastate.edu/cpre288
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 their 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
  
```c
if (str && *str != '\0')
...
```

http://class.ece.iastate.edu/cpre288
Remember, conditions are evaluated on the basis of zero and non-zero.

The quantity 0x80 is non-zero and therefore TRUE.

```c
if (3 || 6)
    True or False?
```
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++;
}
Switch statement

```c
switch (n) {
    case 15:
    case 17:
        x = 0;
        break;
    case 32:
        x = 1;
        break;
    default:
        x = 2;
}
```

Equivalent if/else if/ else

```c
if (n == 17 || n == 15) {
    x = 0;
} else if (n == 32) {
    x = 1;
} else {
    x = 2;
}
```
Control in C: Switch statement

- Benefit over if/else if/else
  - Compiler creates a binary tree of the cases, which reduces the number of jumps
  - Increases code readability
  - Allows falling through cases if the `break` is omitted for a case
// Syntax

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

http://class.ece.iastate.edu/cpre288
Control Flow in C: For loop

// Syntax

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

Note the use of semicolons
// 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.
// Equivalent loop with bad style

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

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

Example: calculate the sum of an array

```c
for (i = 0, sum = 0; i < N; i++) {
    sum += X[i];
}
```
// Syntax

while (condition)  {
    // loop body
}

http://class.ece.iastate.edu/cpre288
Control Flow in C: While loop

While loop
Example: calculate the length of a string

```c
int strlen(char *s) {
    int n = 0;            // string length

    while (s[n]) {
        n++;
    }

    return n;
}
```
// Syntax

do {
    // loop body
} while (condition);
Do-while loop

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

```c
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
**Continue statement**: Start the next iteration of loop

```c
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.

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