CprE 288 – Introduction to Embedded Systems

Instructors:

Dr. Zhao Zhang (Sections A, B, C, D, E)

Dr. Phillip Jones (Sections F, G, J)

Overview of Today’s Lecture

- Announcements
- Scope
- Memory layout
- Recursive Function
- Interrupts
- Function Pointers
- C Library functions
- Casting

Announcements

- Homework due in class Thursday
- Exam 1, Thursday of next week 9/27

SCOPE

Variable scope

Global vs. Local

Global variable

- Declared outside of all functions
- May be initialized upon program startup
- Visible and usable everywhere from .c file

What happens when local/global have the same name?

- Local takes precedence

Summary

- Local – declared inside of a function, visible only to function
- Global – declared outside all functions, visible to all functions

Variable scope

What happens when you want a local variable to stick around but do not want to use a global variable?

Create a *static* variable

Syntax:

static Type Name;

Static variables are initialized once

- Think of static variables as a **“local” global**
- Sticks around (has persistence) but only the function can access it

Variable scope

Visibility scope: Where a variable is visible

```
int m;

int any_func()
{
   int m;
   m = n = 5;
}
```

Variable scope

C global variable (visible to all program files)

```
int global_var;
```

C file-wide static variables (visible only in this file)

```
static int static_var;
```

Local static variables

```
any_func()
{
    static int static_var;
    …
}
```

Variable scope

Example: How to define and use global variables

In header file myvar.h

```
extern int global_var;
```

In program file myvar.c

```
#include “myvar.h”
int global_var;
```

In program file usevar.c

```
#include “myvar.h”
… /* use myvar */
```

Visibility Scope Across Multiple Files

File1.c

```
// global variable
int count = 0;
```

This instance of “count” is visible in all files in the same project.

File2.c

```
extern int count;
int x = count;
```

This is how to use the global variable “count” declared in file1.c.

“extern” declaration is usually put in a header file.

Visibility Scope Across Multiple Files

File1.c

```
// global variable

int count = 0;
```

Another scenario: We want to use the same name “count” in multiple program files, each as a unique variable instance.

File2.c

```
// another global variable
// with the same name
int count = 100;
```

Bad use. The compiler/linker will report conflicting use of name “count”.

Some complier may tolerate it – still bad practice.

Visibility Scope Across Multiple Files

File1.c

```
// static global variable
static int count = 0;
```

Outside the functions, “static” means to limit the visibility of “count” to this program file only.

“static” is a also a storage class modifier (see later).

File2.c

```
// count for file2.c
static int count = 100;
```

“file2.c” gets its own “count”. There is no conflict.

Each instance of “count” is visible in its own file, not visible in any other file.

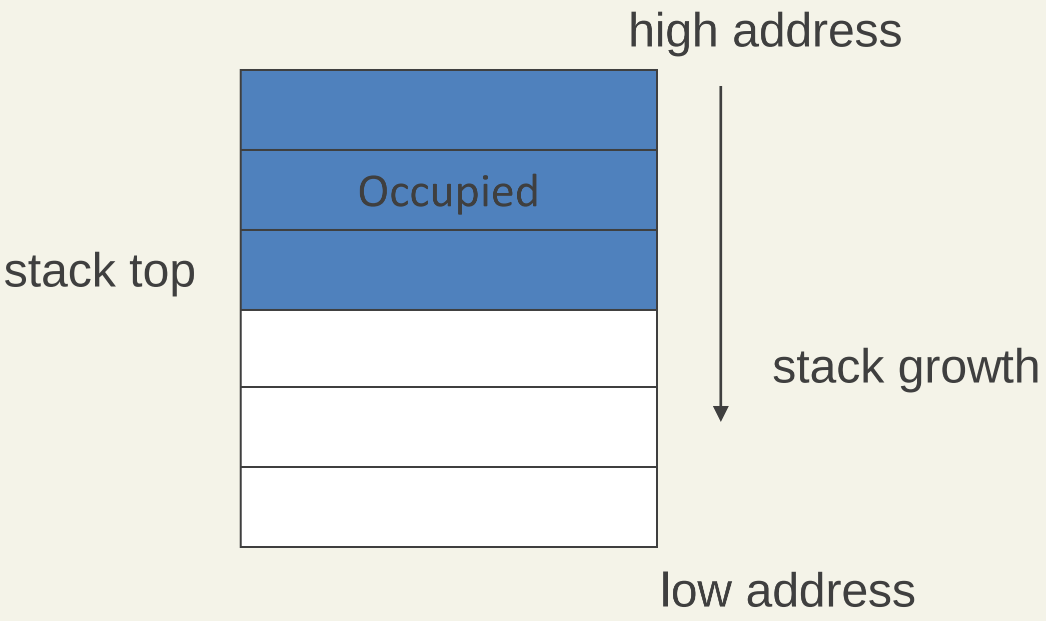
MEMORY LAYOUT

Understanding Data

- Stack
  - Stores data related to function variables, function calls, parameters, return variables, etc.
  - Data on the stack can go “out of scope”, and is then automatically deallocated
  - Starts at the top of the program’s data memory space, and addresses move down as more variables are allocated
- Heap
  - Stores dynamically allocated data
  - Dynamically allocated data usually calls the functions *alloc* or *malloc* (or uses *new* in C++) to allocate memory, and *free* to (or *delete* in C++) deallocate
  - There’s no garbage collector!
  - Starts at bottom of program’s data memory space, and addresses move up as more variables are allocated

Function and Stack

Conventional program stack grows downwards: New items are put at the top, and the top grows down



Function and Stack

Auto, local variables have their storage in stack

Why stack?

- The LIFO order matches perfectly with functions call/return order
  - LIFO: Last In, First Out
  - Function: Last called, first returned
- Efficient memory allocation and de-allocation
  - Allocation: Decrease SP (stack top)
  - De-allocation: Increase SP

Function and Stack

Function Frame: Local storage for a function

Example: 1. A is called; 2. A calls B; 3. B calls C; 4. C returns

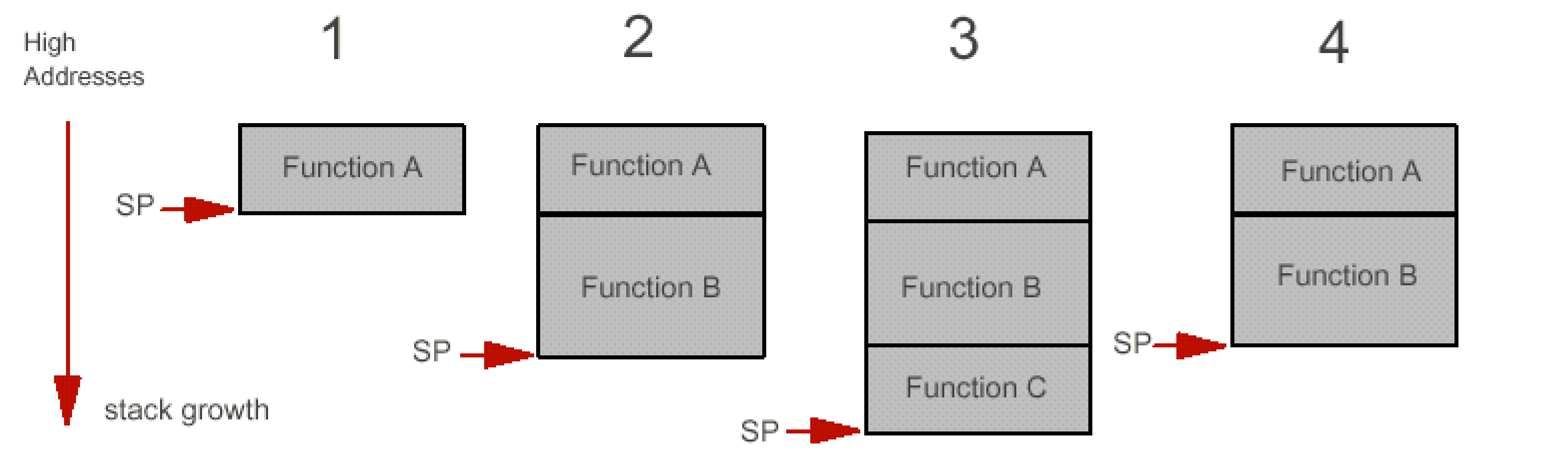


Figure 1 - Stack Frame creation and destruction

Function and Stack

What can put in a stack frame?

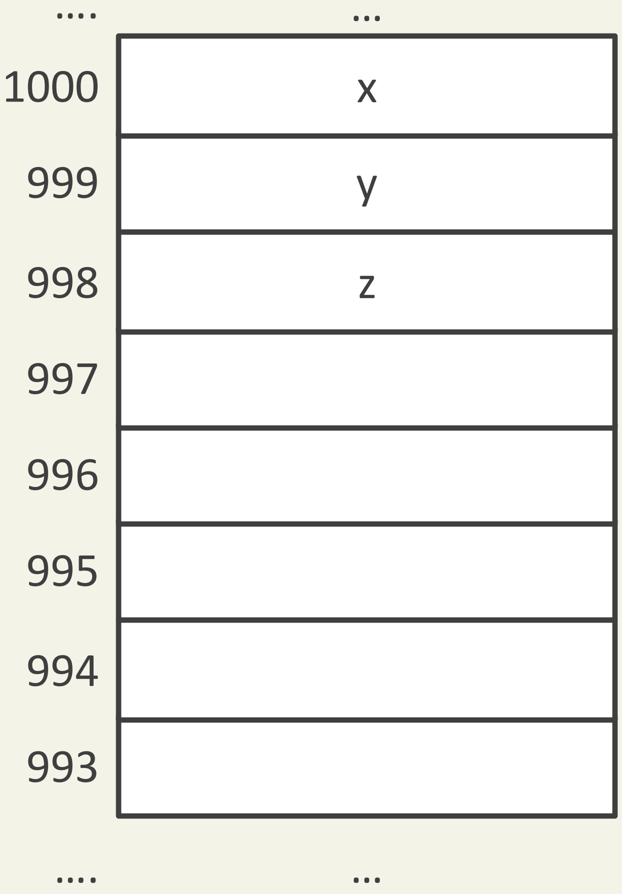
- Function return address
- Parameter values
- Return value
- Local variables
- Saved register values

Example: Stack

- The following example shows the execution of a simple program (left) and the memory map of the stack (right)

Example: Stack

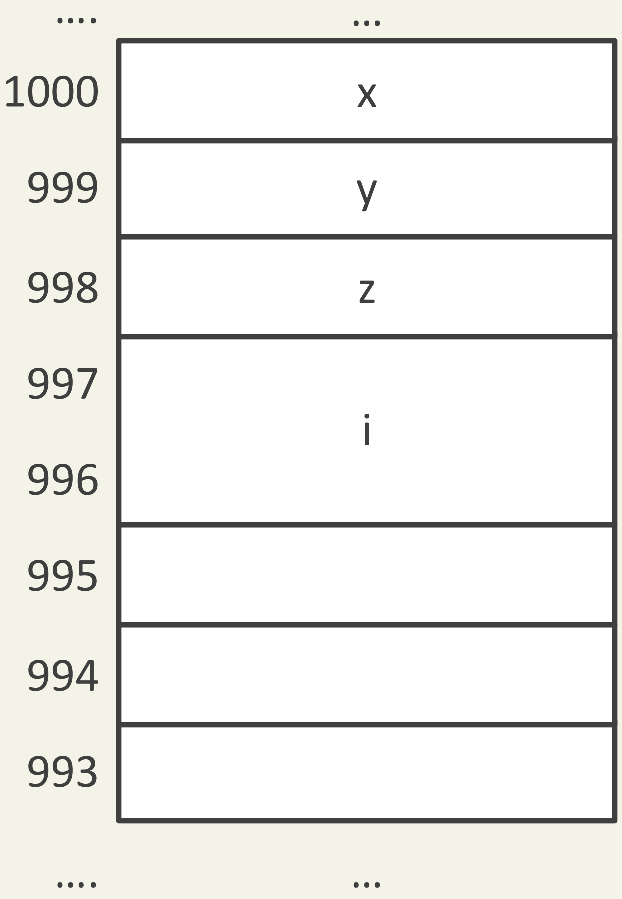
```
void doNothing() {
        char c;
}

int main() {
        char x, y, z;
        int i;
        for (i = 0; i < 10; i++) {
                doNothing();
        }
        return 0;
}
```



….	…
1000	x
999	y
998	z
997	
996	
995	
994	
993	
….	…

Example: Stack

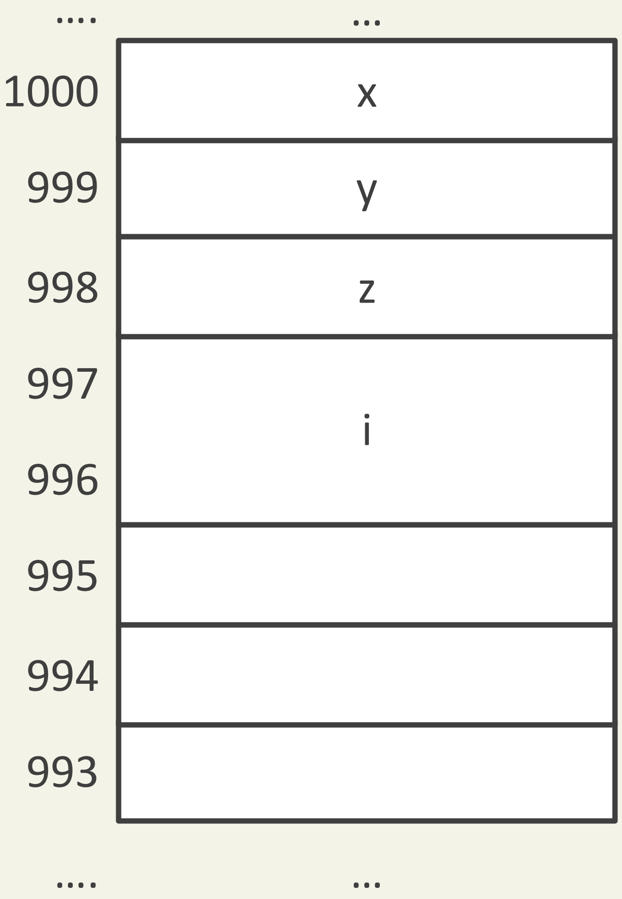
```
void doNothing() {
        char c;
}

int main() {
        char x, y, z;
        int i;
        for (i = 0; i < 10; i++) {
                doNothing();
        }
        return 0;
}
```



….	…
1000	x
999	y
998	z
997	i
996	
995	
994	
993	
….	…

Example: Stack

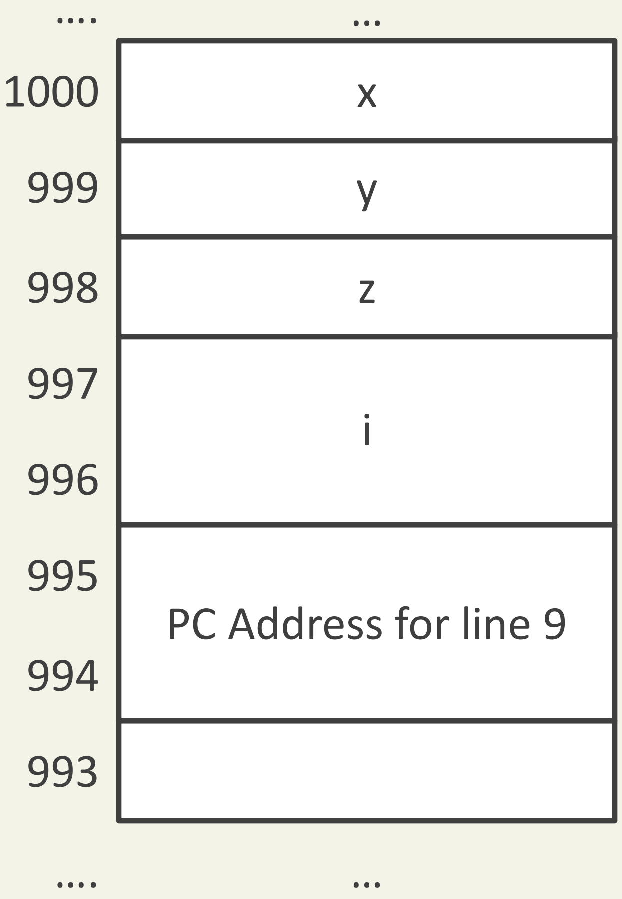
```
void doNothing() {
        char c;
}

int main() {
        char x, y, z;
        int i;
        for (i = 0; i < 10; i++) {
                doNothing();
        }
        return 0;
}
```



….	…
1000	x
999	y
998	z
997	i
996	
995	
994	
993	
….	…

Example: Stack

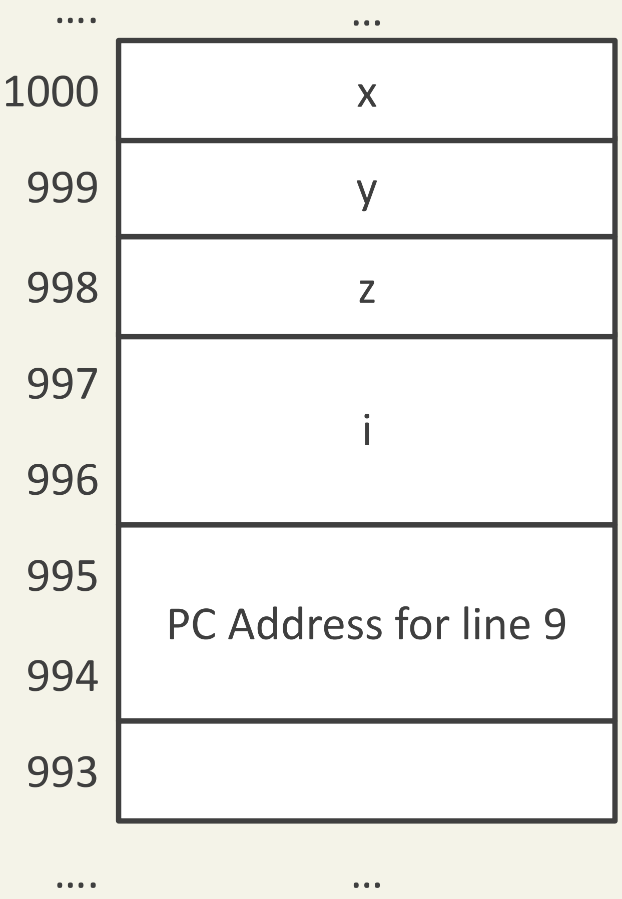
```
void doNothing() {
        char c;
}

int main() {
        char x, y, z;
        int i;
        for (i = 0; i < 10; i++) {
                doNothing();
        }
        return 0;
}
```



....	...
1000	x
999	y
998	z
997	i
996	
995	PC Address for line 9
994	
993	
....	...

Example: Stack

```
void doNothing() {
        char c;
}

int main() {
        char x, y, z;
        int i;
        for (i = 0; i < 10; i++) {
                doNothing();
        }
        return 0;
}
```



….	…
1000	x
999	y
998	z
997	i
996	
995	PC Address for line 9
994	
993	
….	…

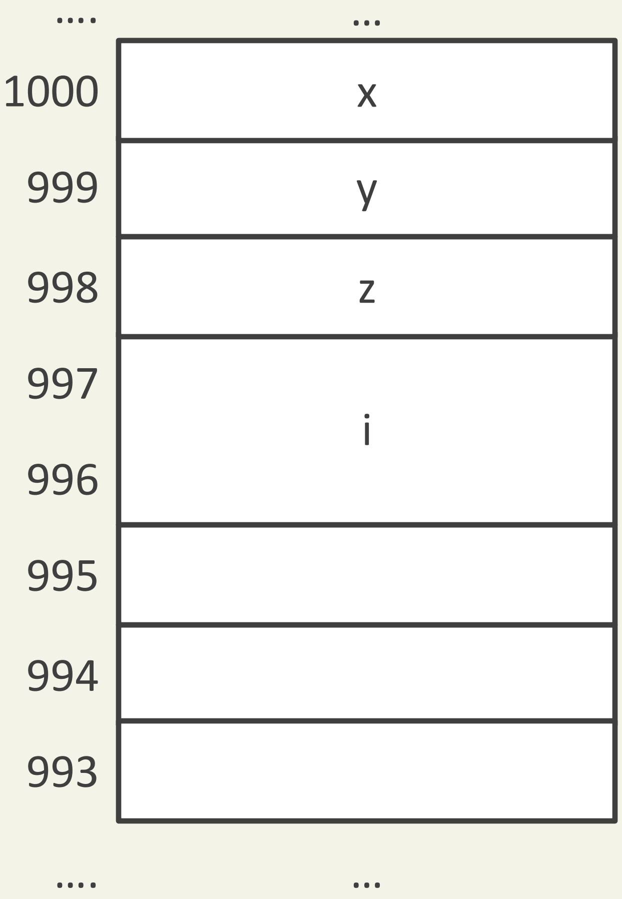
Example: Stack

```
void doNothing() {
        char c;
}

int main() {
        char x, y, z;
        int i;
        for (i = 0; i < 10; i++) {
                doNothing();
        }
        return 0;
}
```

….	…
1000	x
999	y
998	z
997	i
996	
995	PC Address for line 9
994	
993	c
….	…

Example: Stack

```
void doNothing() {
        char c;
}

int main() {
        char x, y, z;
        int i;
        for (i = 0; i < 10; i++) {
                doNothing();
        }
        return 0;
}
```



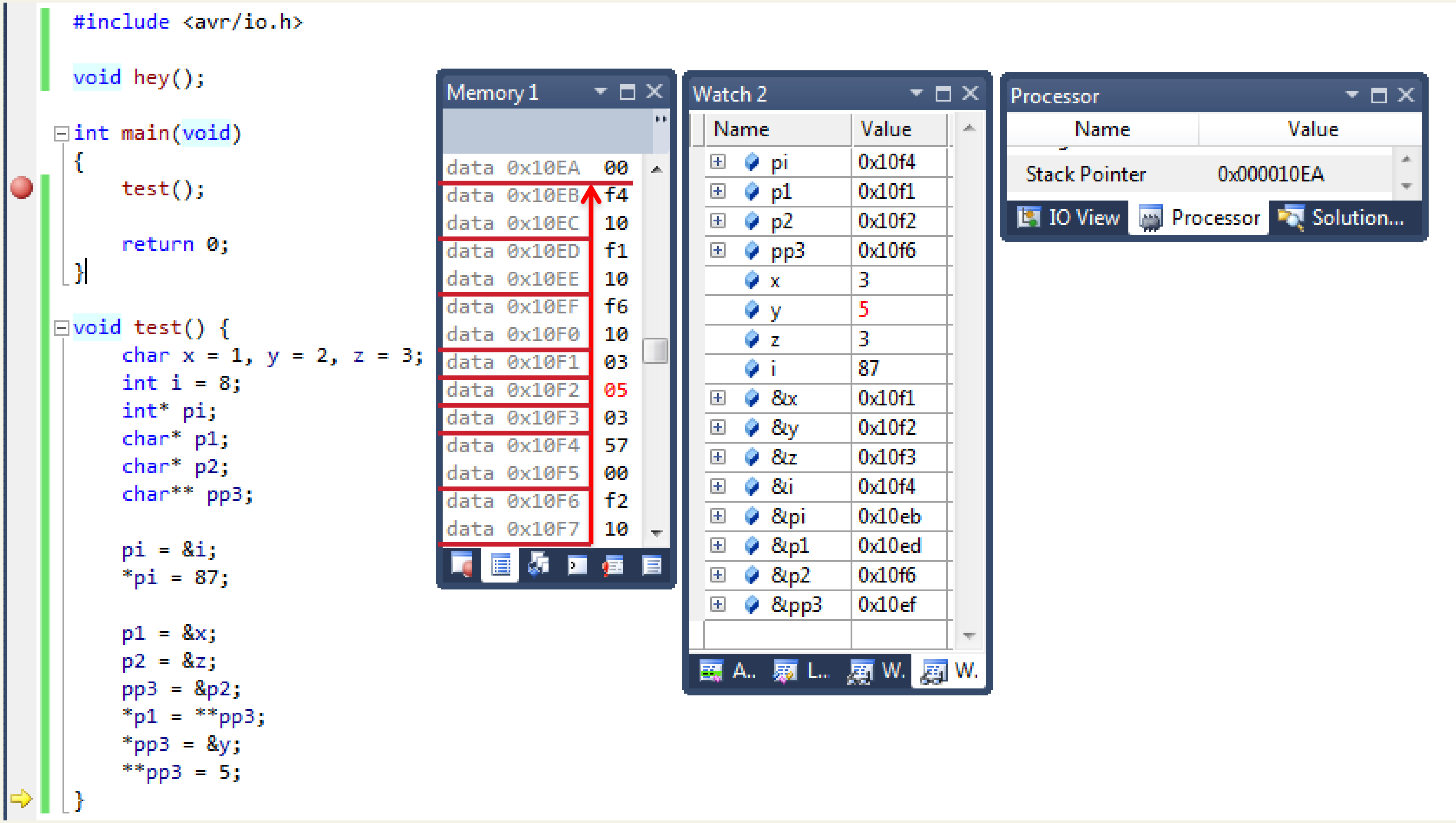
….	…
1000	x
999	y
998	z
997	i
996	
995	
994	
993	
….	…

Stack Memory Layout: Example

```
char x = 1, y = 2, z = 3;
int i = 8;
int* pi;
char* p1;
char* p2;
char** pp3;

pi = &i;
*pi = 87;          // i = 87;

p1 = &x;
p2 = &z;
pp3 = &p2;
*p1 = **pp3;       // x = z;
*pp3 = &y;
**pp3 = 5;         // y = 5;
```

- Class work out on board. Final values for all memory locations.

Stack Memory Layout: Example



```
#include <avr/io.h>

void hey();

int main(void)
{
    test();

    return 0;
}

void test() {
    char x = 1, y = 2, z = 3;
    int i = 8;
    int* pi;
    char* p1;
    char* p2;
    char** pp3;

    pi = &i;
    *pi = 87;

    p1 = &x;
    p2 = &z;
    pp3 = &p2;
    *p1 = **pp3;
    *pp3 = &y;
    **pp3 = 5;
}
```

```
data 0x10EA  00
data 0x10EB  f4
data 0x10EC  10
data 0x10ED  f1
data 0x10EE  10
data 0x10EF  f6
data 0x10F0  10
data 0x10F1  03
data 0x10F2  05
data 0x10F3  03
data 0x10F4  57
data 0x10F5  00
data 0x10F6  f2
data 0x10F7  10
```

Name	Value
pi	0x10f4
p1	0x10f1
p2	0x10f2
pp3	0x10f6
x	3
y	5
z	3
i	87
&x	0x10f1
&y	0x10f2
&z	0x10f3
&i	0x10f4
&pi	0x10eb
&p1	0x10ed
&p2	0x10f6
&pp3	0x10ef

Name	Value
Stack Pointer	0x000010EA

Note: Before calling test(), the stack pointer started at 0x10FB, added the program counter and the current stack pointer to the stack (at address 0x10F9 and 0x10FB)

Memory Address Space

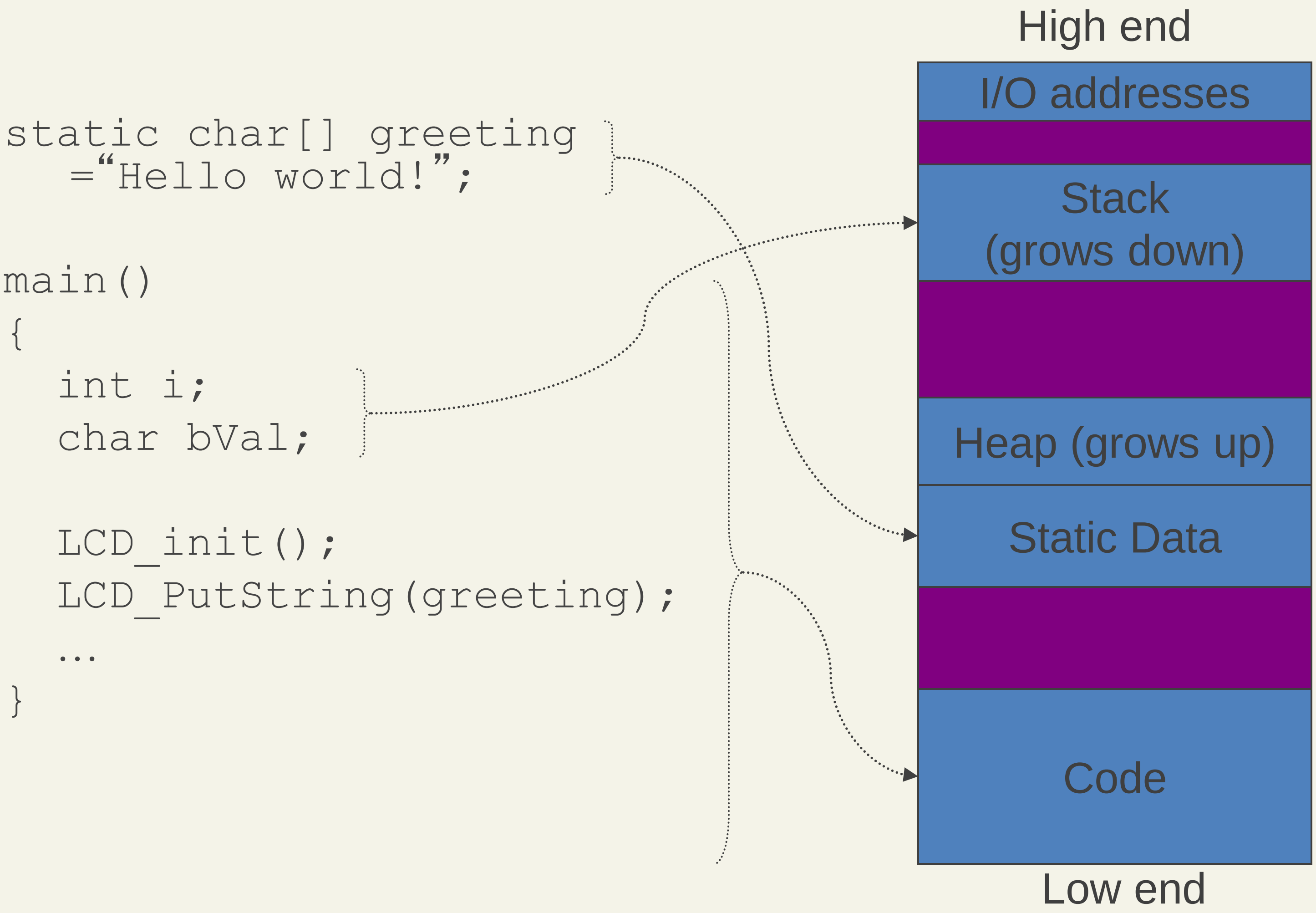
It is the addressability of the memory

- Upper bound of memory that can be accessed by a program
- The larger the space, the more bits in memory addresses
- 32-bit address – accessibility to 4GB memory

What are

- Virtual memory address space
- Physical memory address space
- Physical memory size
- I/O addresses (ports)

General Memory Layout



```
static char[] greeting
   =“Hello world!”;

main()
{
   int i;
   char bVal;

   LCD_init();
   LCD_PutString(greeting);
   …
}
```

High end

I/O addresses

Stack
(grows down)

Heap (grows up)

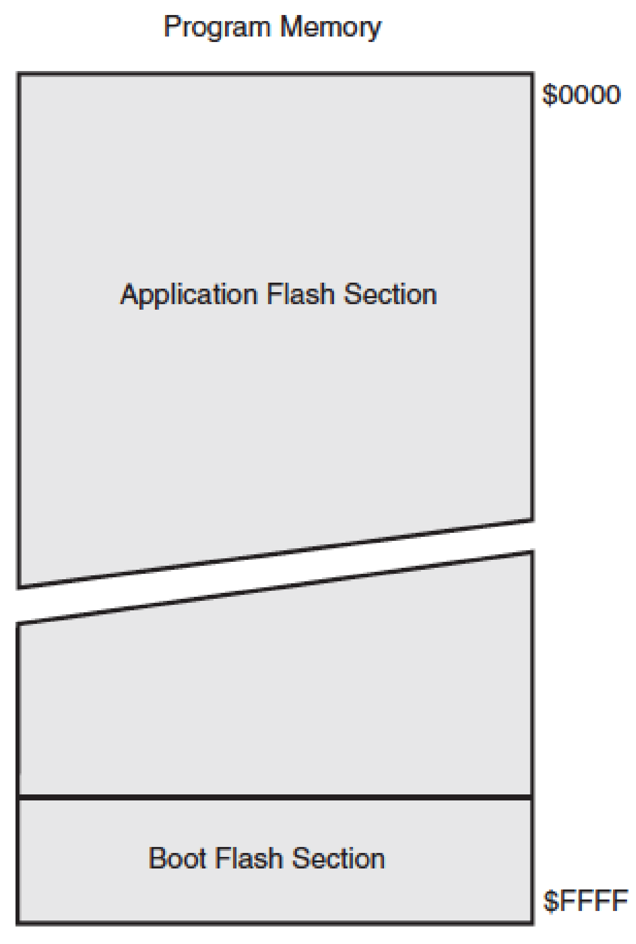
Static Data

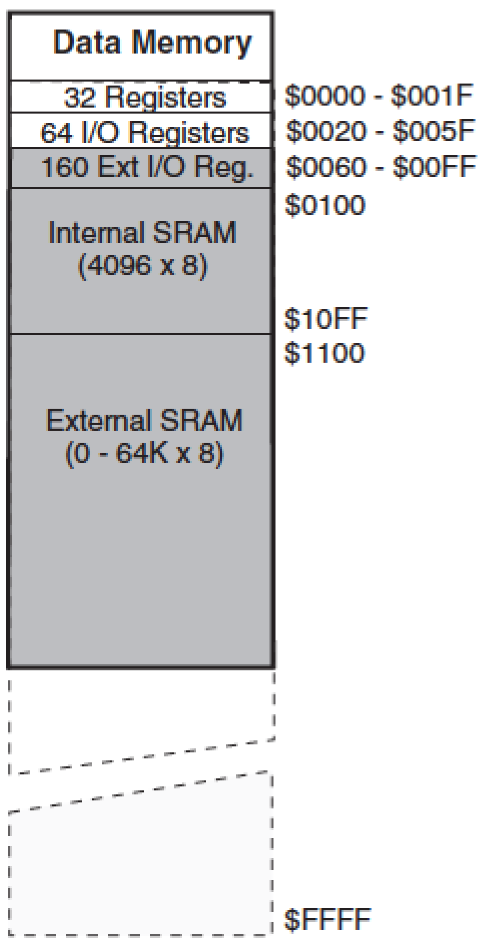
Code

Low end

ATmega128 Memory Layout

Harvard Architecture: Two separate memory address spaces for instruction and data





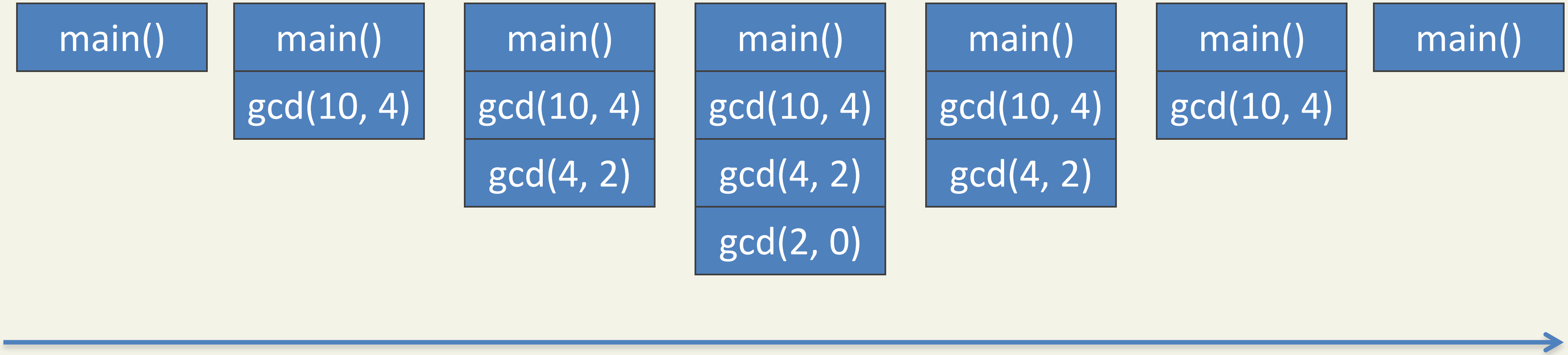
Recursive Function

A function that calls itself

```
/* calculate the greatest common
    divisor */
int gcd(int m, int n)
{
    if (n == 0)
        return m;
    else
        return gcd(n, m % n);
}
```

Function and Stack

The use of stack by a recursive function:



main()

main()
gcd(10, 4)

main()
gcd(10, 4)
gcd(4, 2)

main()
gcd(10, 4)
gcd(4, 2)
gcd(2, 0)

main()
gcd(10, 4)
gcd(4, 2)

main()
gcd(10, 4)

main()

What happens if a function keeps calling itself and does not end the recursion?

ISR (INTERRUPT SERVICE ROUTINES)

Interrupt Service Routine

Interrupt: Hardware may raise interrupt to inform the CPU exceptional events

- Timer expires
- ADC gets a new datum
- A network packet arrives

Conceptually, it' <u>like</u> the CPU calls your ISR function

- You will learn more low-level details when studying assembly
- ISR: Interrupt Service Rutine

Interrupt Service Routine

ISR is a function that runs when there is an interrupt from a internal or external source

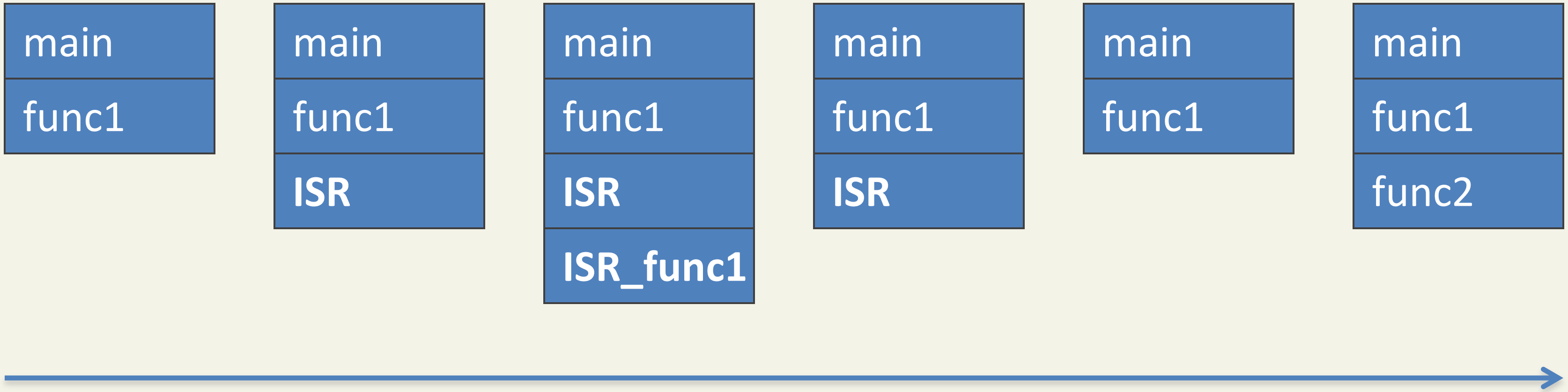
1. An interrupt occurs
2. Foreground program is suspended
3. The ISR is executed
4. Forgound program is resumed

An ISR is a special type of function

- No return value and no parameters

Interrupt Service Routine

Example of stack use in ISR execution:



An ISR function saves register context (to be studied), may call other functions, and restore register context and stack top before it returns.

ISR Example: Lab 4

```
int main()
{
        lcd_init();
        timer_init();  // enable interrupt
        while (1) {
            // do nothing
        }
}
```

ISR Example: Lab 4

```
/* Timer interrupt source 1: the function will be
   called every one second to update clock */
ISR (TIMER1_COMPA_vect)
{
   // YOUR CODE
}

/* Timer interrupt source 2: for checking push
 button five times per second*/
ISR (TIMER3_COMPA_vect)
{
   // YOUR CODE
}
```

An ISR Macro automatically associate the ISR function with an interrupt source

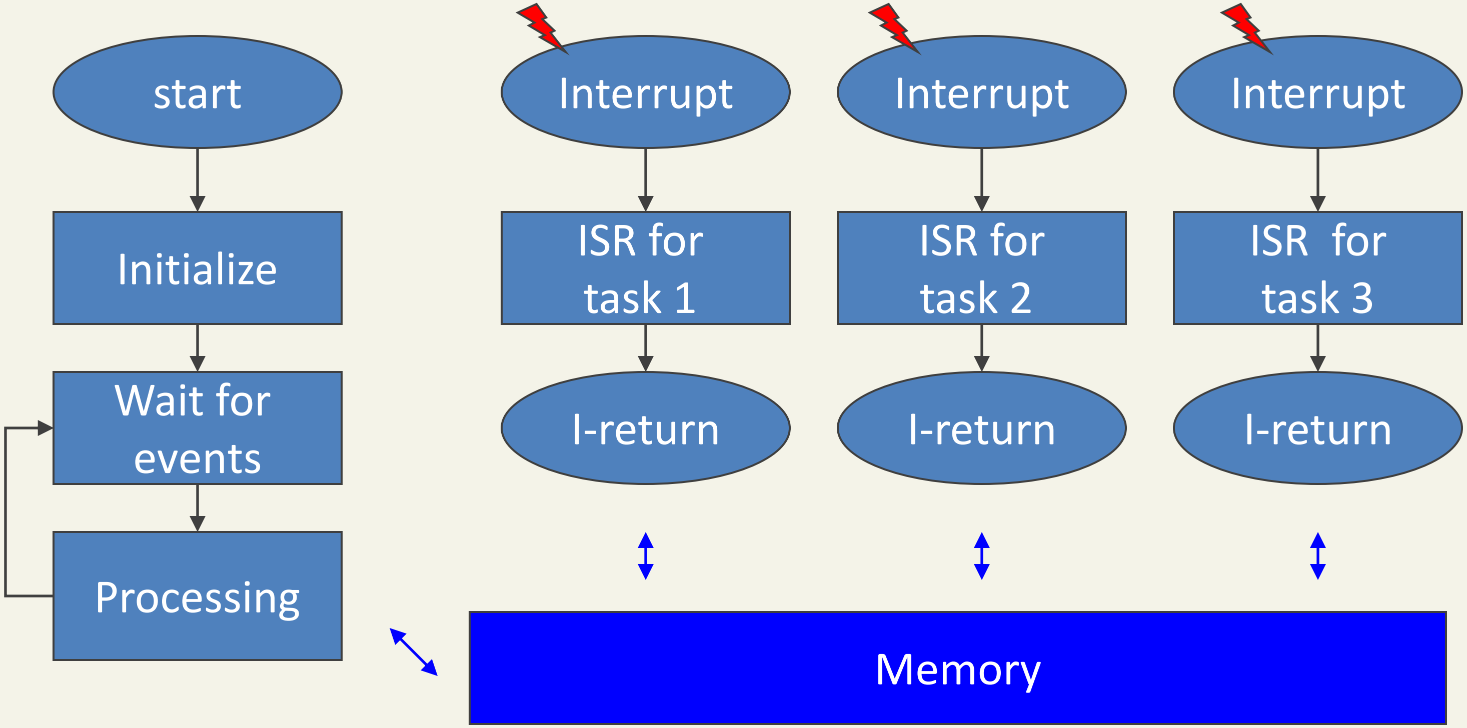
- **`TIMER1_COMPA_vect`**: ATMega128 Timer 1 Output Compare A match (to be studied)
- **`TIMER3_COMPA_vect`**: ATMega128 Timer 3 Output Compare A match

Volatile Varaibles

Volatile variable: The memory content may change even if the running code doesn’t change it.

```
volatile unsigned char pushbutton_reading;

ISR (TIMER3_COMPA_vect)
{
  … // read PORT for push button
  pushbutton_reading = …;
}

main()
{
  while (!pushbutton_reading)
    {}
  … // other code
}
```

Interrupt in Embedded Systems



Adapted from fundaments of embedded software, fig 7-1

ISR Macro

- Two easy steps to using interrupts

1. Enable the interrupt (every interrupt has an enable bit)
   - Look up in the datasheet to see what register name and bit position you will need to set.
2. Write the ISR (interrupt service routine)
   - The ISR is a function, or block of code, that the processor will call for you whenever the interrupt event occurs
   - The ISR macro needs one parameter: the name of your interrupt vector. You can find a list of interrupt vectors here: http://www.nongnu.org/avr-libc/user-manual/group__avr__interrupts.html

FUNCTION POINTERS

Function Pointer

A pointer to function

- Call a function through a pointer variable
- More efficient than using if- or switch-statement
- Also used to implement virtual functions (e.g. in C++ and Java)

Why does it work?

- A C function becomes a block of binary machine instructions after compilation
- Each function has a starting address; a function call is to make a jump to the starting address
- The starting address can also be stored into a variable, and a jump can be made by loading the address into PC (program counter)

Function Pointer

Example: Dynamically set the right function to call

```
int quickSort(int X[], int size);
int mergeSort(int X[], int size);

int X[] = {1, 2, 3, …};
int N = …;

main()
{
   int (*mySort)(int X[], int size);

   if (…)       // some condition
      mySort = quickSort;
   else
      mySort = mergeSort;

   // can also be (*mySort)(X, N)
   mySort(X, N);
}
```

Function Pointer

Example: Dynamically set the right function to call

```
int quickSort(int X[], int size);
int mergeSort(int X[], int size);

int X[] = {1, 2, 3, …};
int N = …;

main()
{
  int (*mySort)(int X[], int size);

  if (…)       // some condition
    mySort = quickSort;
  else
    mySort = mergeSort;

  // can also be (*mySort)(X, N)
  mySort(X, N);
}
```

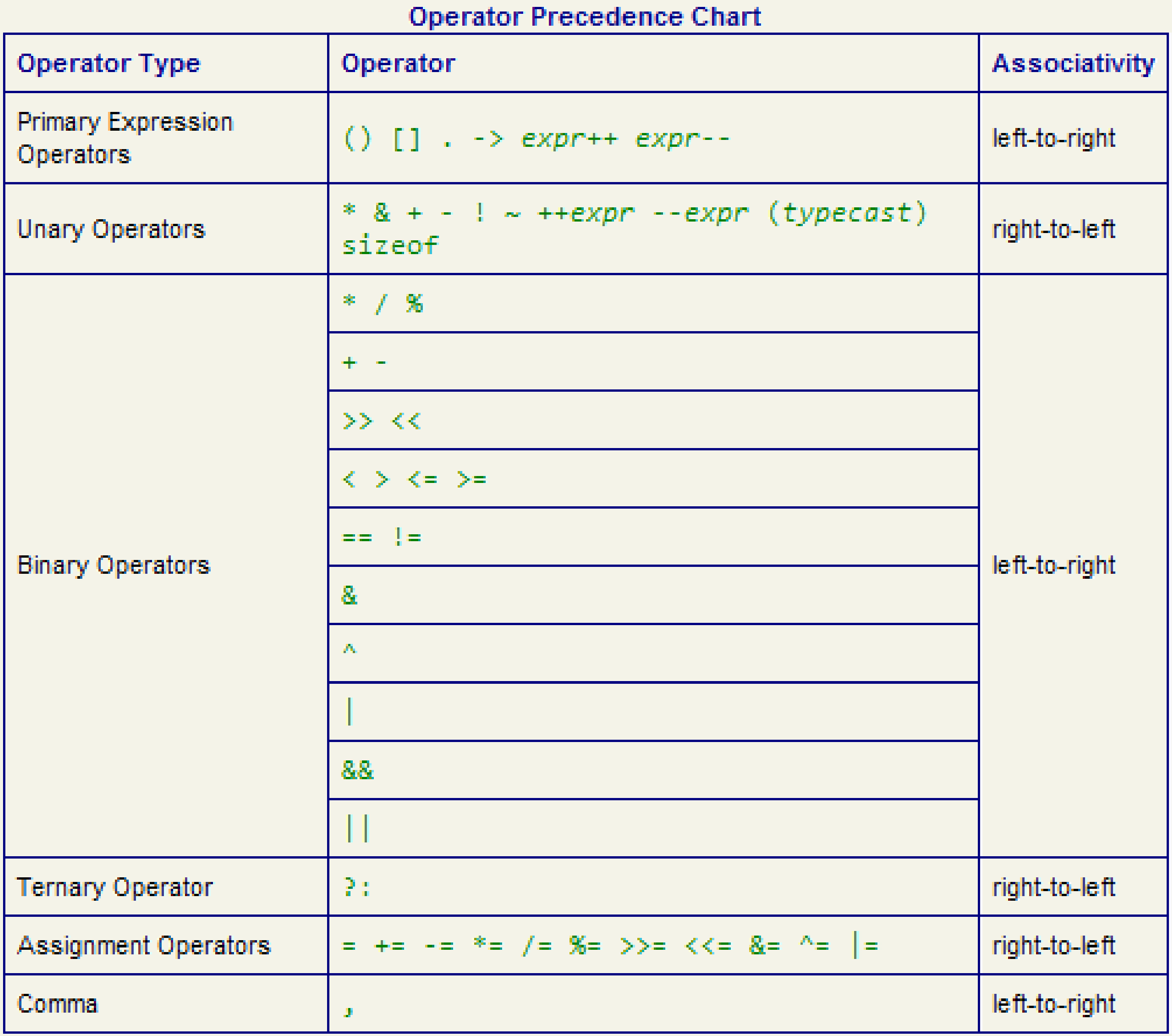
Function Pointer

Every function has a starting address – that's its value in C

Print out the address of main()

```
printf (“%x\n”, main);
```

OPERATOR PRECEDENCE



Operator Precedence Chart

Operator Type	Operator	Associativity
Primary Expression Operators	`() [] . -> expr++ expr--`	left-to-right
Unary Operators	`* & + - ! ~ ++expr --expr (typecast) sizeof`	right-to-left
Binary Operators	`* / %`	left-to-right
	`+ -`	
	`>> <<`	
	`< > <= >=`	
	`== !=`	
	`&`	
	`^`	
	`|`	
	`&&`	
	`||`	
Ternary Operator	`?:`	right-to-left
Assignment Operators	`= += -= *= /= %= >>= <<= &= ^= |=`	right-to-left
Comma	`,`	left-to-right

Exercise: Operation Precedence

```
a*b + c*d          same as(a*b) + (c*d)
```

How about the following expression and condition?

```
    x + y * z + k                          x + (y * z) + k

    *str++                                 *(str)
                                           str = str + 1;

    if (a == 10 && b == 20)                if ((a == 10) && (b == 20))

    if (a & 0x0F == b & 0x0F)              if (a & (0x0F == b) & 0x0F)

    if ((a & 1) == 0)
```

Are ()'s required?

```
x & (0x10 == 0x10)
```

```
x & (!y)
```

```
(x == 23) && (y < 12)
```

```
int array[50] = {1, 2, 3, 4, -1};
do {
    (*array)++;
} while (*array++);
```

TYPE CONVERSION (CASTING)

Type Conversion and Casting

Recall C has the following basic data types:

```
char, short, int, long, float, double
```

Assume:

```
char c;   short h;   int n;   long l;
float f; double d;
```

What’s the meaning of

```
c = h;
n = h;
f = n;
(f > d)
```

Implicit Conversion

A longer integer value is cut short when assigned to a shorter integer variable or char variable

```
char c;

short h = 257;

long l;


c = h;          // The rightmost 8-bit of h is copied into c


n = l;          // The rightmost 16-bit of l is copied into n
```

Implicit Conversion

A shorter integer value is extended before being assigned to a longer integer variable

```
l = h;          // the 16-bit value of h is extended to 32-bit

h = c;          // the 8-bit value of c is extended to 16-bit
                // signed extension or not is dependent on
                //     the system
```

Implicit Conversion

A double type is converted to float type and vice versa using IEEE floating point standard

```
d = 10.0;      // 10.0 with double precision
f = d;         // 10.0 with single precision

f = 20.0;      // 20.0 with single precision
d = f;         // 20.0 with double presion
```

Implicit Conversion

A float/double is floored to the closest integer when assigned to an integer/char variable

```
f = 10.5;
n = f;          // n = 10

d = -20.5;
l = d;          // l = -20
```

Implicit Conversion

In an expression:

- A shorter value is converted to a longer value before the operation
- The expression has the type of the longer one

(c + h) c is extended to 16-bit and then added with h

(n + l) n is extended to 32-bit and then added with l

(f + d) f is extended to double precision before being added with d

Implicit Conversion

A float/double is floored to the closest integer when assigned to an integer/char variable

```
f = 10.5;
n = f;          // n = 10

d = -20.5;
l = d;          // l = -20
```

Explicit Conversion: From String to Others

```
#include <inttype.h>
#include <stdlib.h>

n = strtol(“10”);           // n = 10
f = strtof(“2.5”);          // f = 2.5 in single precision
d = strtod(“2.5”);          // d = 2.5 in double precision
```

strtol: string to long

strtof: string to float

strtod: string to double

Explicit Casting

```
int i = 60;
float f = 2.5;

f = (float) (i + 3);
```

Type Casting

Explicitly convert one data type to another data type

(type name) expression

```
int n1 = -1;
unsigned int n2 = 1;

if (n1 < (int) n2)                    // this is true

if ((unsigned int) n1 < n2)                 // this is false
```

C LIBRAY FUNCTIONS

C Library Functions

In C many things are carried out by library functions

- Simple language, rich libraries

Commonly used libraries

- File I/O (include user input/output)
- String manipulations
- Mathematical functions
- Process management
- Networking

C Library Functions

Use standard file I/O

```
/* include the header file for I/O lib */
#include <stdio.h>

main()
{
  /* use the fprintf function */
  fprintf(stdout, “%s\n”, “Hello World\n”);
}
```

C Library Functions

Formatted output: printf, fprintf, sprintf and more; use conversion specifiers as follows

%s	string
%d	signed decimal
%u	unsigned decimal
%x	hex
%f	floating point (float or double)

How to output the following variables in format “a = …, b =…, c = …, str = …” in a single line?

```
int a;
float b;
int *c;
char str[10];
```

C Library Functions

String operations: copy, compare, parse strings and more

```
#include <string.h>
```

- strcpy: copy one string to another
- strcmp: compare two strings
- strlen: calculate the length of a string
- strstr: search a string for the occurrence of another string

C Library Functions

Error processing and reporting: use exit function

```
#include <stdio.h>
#include <stdlib.h>
...
void myfunc(int x)
{
  if (x < 0) {
    fprintf(stderr, “%s\n”,
          “x is out of range”);
    exit(-1);
  }
}
```

C Library Functions

Math library functions

```
#include <math.h>
...
   n = round (x); /* FP round function */
...
```

To build:

```
gcc –Wall –o myprogram –lm myprogram.c
```

C Library Functions

How to find more?

On Linux machines: Use man

```
man printf
man string
man string.h
man math.h
```

Most functions are available on Atmel platform

C Library Functions

More information on C Library functions:http://www.acm.uiuc.edu/webmonkeys/book/c_guide/

Other commonly used:

- stdlib.h: Some general functions and macros
- assert.h: Run-time self checking
- ctype.h: Testing and converting char values

C Library Functions

AVR Libc Home Page: http://www.nongnu.org/avr-libc/

Non AVR-specific:

- alloca.h: Allocate space in the stack
- assert.h: Diagnostics
- ctype.h: Character Operations
- errno.h: System Errors
- inttypes.h: Integer Type conversions
- math.h: Mathematics
- setjmp.h: Non-local goto
- stdint.h: Standard Integer Types
- stdio.h: Standard IO facilities
- stdlib.h: General utilities
- string.h: Strings

C Library Functions

AVR Libc Home Page: http://www.nongnu.org/avr-libc/

AVR-specific

- avr/interrupt.h: Interrupts
- avr/io.h: AVR device-specific IO definitions
- avr/power.h: Power Reduction Management
- avr/sleep.h: Power Management and Sleep Modes
- util/setbaud.h: Helper macros for baud rate calculations
- Many others