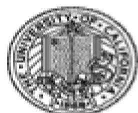
CMPE-013/L

Introduction to "C" Programming

Max Lichtenstein

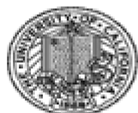
based on slides from Maxwell James Dunne



What is it counting?

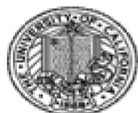
```c
int count=0;
while(x)
{
    count++;
    x = x&(x-1);
}
return count;
```

```
li $counter, 0
li $x 0xDEADBEEF
while_x_loop:                           # while (x != 0) {
        beqz $x exit_loop
        addi $counter, $counter, 1          # counter++;
        subi $t2, $x, 1                     # $t2 = x-1
        and $x, $x, $t2                     # x = x & (x-1)
        b while_x_loop              # }
exit_loop: nop
```



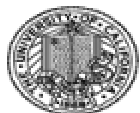
What does it print?

```
float f = 0.0;
float one = 1.0;
int i;

for (i = 0; i < 10; i++) {
    f = f + 0.1;
}

if (f == one){
    printf("f is 1.0\n");
} else {
    printf("f is NOT 1.0\n");
}
```



Roadmap

- Lab 1
- Compiling vs Interpreting review
- Fundamental Elements of C
  - Preprocessing (Comments, macros)
  - Variables + datatypes
  - Control structures
  - Functions (?)
  - Scope (?)
- Lab 2 preview?

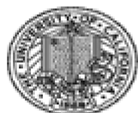


Lab01_submission_07

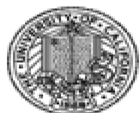
lower

Office Hours

- Max's Office Hours:
  - W 3pm - 4pm, (E2 316)
  - F 1pm - 2pm (E2 316)
- Pavlo's Office Hours:
  - M 2pm - 4pm (Soc Sci 1, 135, tentative)
  - T 2pm - 4pm (location TBD)

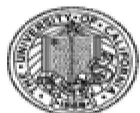


Tutor: Natalie King



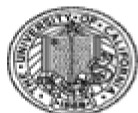
Lab 1

- How did it go?
- 1 day extension



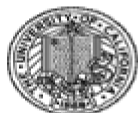
C: A High Level Programming Language

- Gives symbolic names to values
  - Don’t need to know which register or memory location
    - *(usually)*
- Provides abstraction of underlying hardware
  - operations do not depend on instruction set
  - example: can write “a = b * c”, even though underlying hardware may not have a multiply instruction



C: A High Level Programming Language

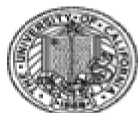
- Provides expressiveness
  - use meaningful symbols that convey meaning
  - simple expressions for common control patterns (if-then-else)
- Enhances code readability
- Safeguards against bugs
  - can enforce rules or conditions at compile-time or run-time



Compilation vs. Interpretation

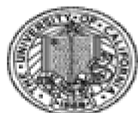
- Different ways of translating high-level language

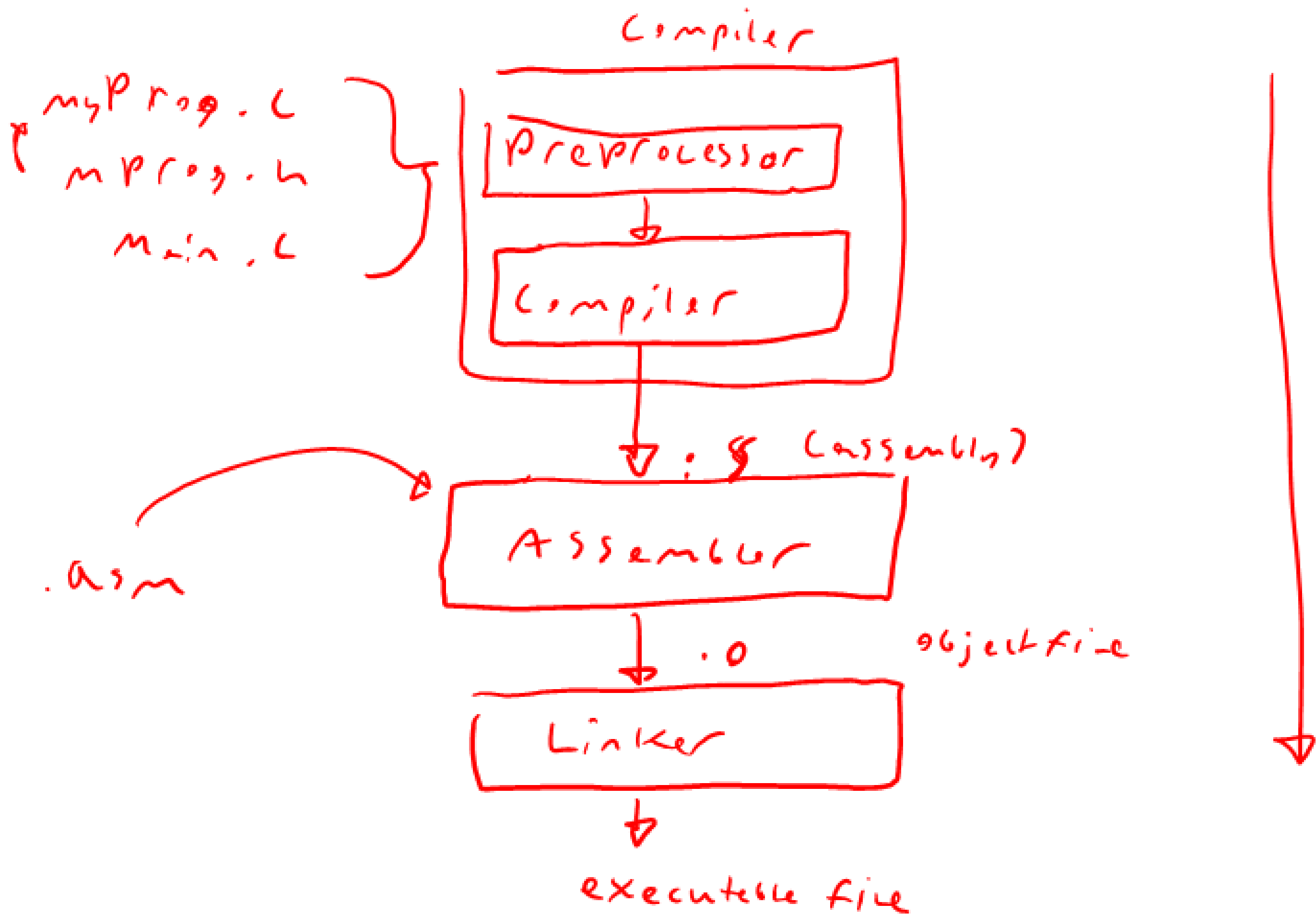
- Interpretation
  - interpreter = program that executes program statements
  - generally one line/command at a time
  - limited processing
  - easy to debug, make changes, view intermediate results
  - languages: BASIC, LISP, Perl, Java, Matlab, Python



Compilation vs. Interpretation

- Compilation
  - translates statements into machine language
  - does not execute, but creates executable program
  - performs optimization over multiple statements
  - change requires recompilation
    - can be harder to debug, since executed code may be different
  - languages: C, C++, Fortran, Pascal, Ada




Compiler
myProg.c
mProg.h
main.c
Preprocessor
Compiler
.s (assembly)
.asm
Assembler
.o
object file
Linker
executable file

Compilation vs. Interpretation

- Consider the following algorithm:

  Read W from the keyboard.

  X = W + W

  Y = X + X

  Z = Y + Y

  Print Z to screen.

- If interpreting, how many arithmetic operations occur?

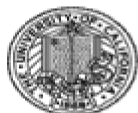
- If compiling, we can analyze the entire program and possibly reduce the number of operations. Can we simplify the above algorithm to use a single arithmetic operation?

$Z = 2 \cdot Y$

$2 \cdot (2x)$

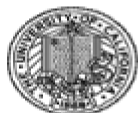
$Z = 2 \cdot 2 \cdot 2$ ✓

$Z = 8 \cdot W$



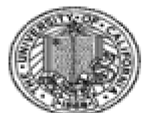
Compilation, Generalized

- Source Code Analysis
  - "front end"
  - parses programs to identify its pieces
  - variables, expressions, statements, functions, etc.
  - depends on language (not on target machine)
- Code Generation
  - "back end"
  - generates machine code from analyzed source
  - may optimize machine code to make it run more efficiently
  - very dependent on target machine
- Symbol Table
  - map between symbolic names and items
  - like assembler, but more kinds of information



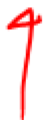
Generic C Code

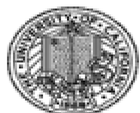
```
#include <stdio.h>

int main(void)
{
        printf("Hello, world!\n"); // Uses the I/O library to print

        return 0;
}
```



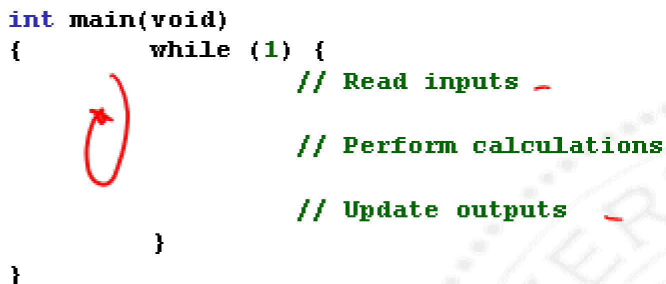
Embedded C Code

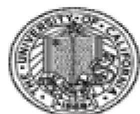
```
#include <stdio.h>
#include <some_hardware_library.h>

int main(void)
{
        InitializePrintHardware();

        printf("Hello, world!\n");

        while (1); // Loop forever and never return
}
```



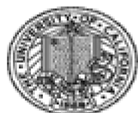


Embedded C Code





Preprocessing



#include Directive

- Three ways to use the #include directive:

Syntax

```
#include <file.h>
```

Look for file in the compiler search path

The compiler search path usually includes the compiler's directory and all of its subdirectories.

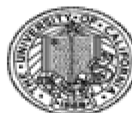
For example: C:\Program Files\Microchip\MPLABX\XC16*.*

```
#include "file.h"
```

Look for file in project directory only

```
#include "c:\MyProject\file.h"
```

Use specific path to find include file



main.c

#include <lib.h>
#include <lib.c>

lib.h

Foo()

Foo(){
}

Foo(){
}

`#include` Directive

main.h Header File and main.c Source File

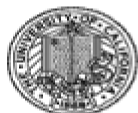
main.h

```c
unsigned int a;
unsigned int b;
unsigned int c;
```

The contents of main.h are *effectively* pasted into main.c starting at the #include directive's line

main.c

```c
#include "main.h"

int main(void)
{
        a = 5;
        b = 2;
        c = a+b;
}
```



#include Directive

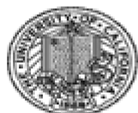
Equivalent main.c File

- After the preprocessor runs, this is how the compiler sees the main.c file
- The contents of the header file aren't *actually* copied to your main source file, but it will behave *as if* they were copied

main.c

```
unsigned int a;
unsigned int b;
unsigned int c;

int main(void)
{
        a = 5;
        b = 2;
        c = a+b;
}
```

Equivalent main.c file without #include



Header Guards

Duplicate #includes

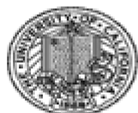
main.h

```c
unsigned int a;
unsigned int b;
unsigned int c;
```

The contents of main.h are *effectively* pasted twice into main.c starting at the #include directive's line

main.c

```c
#include "main.h"
#include "main.h"

int main(void)
{
        a = 5;
        b = 2;
        c = a + b;
}
```



Header guards

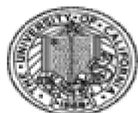
Equivalent main.c File

- Duplicate declarations will occur.
- Which will give compilation errors as there cannot exist multiple declarations of the same variable in the same scope.

main.c

```
unsigned int a;
unsigned int b;
unsigned int c;
unsigned int a;
unsigned int b;
unsigned int c;

int main(void)
{
        ...
}
```

Equivalent main.c file without #include



Header guards

Realistic example

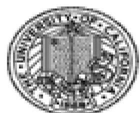
OledDriver.h

```
#include "OledDriver.h"
```

Oled.h

main.c

```
#include "OledDriver.h"
#include "Oled.h"
```



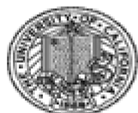
Header guards

How do you write/use them

- Declare a macro when a header file is processed.
- Check for that macro before including the code.

Oled.h

```
#ifndef OLED_H
#define OLED_H

#include "OledDriver.h";

...

#endif // OLED_H
```



orig main

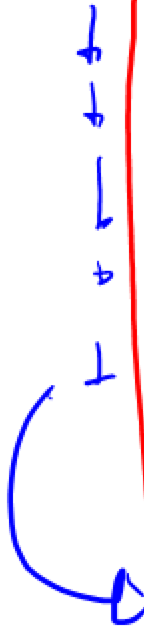
main

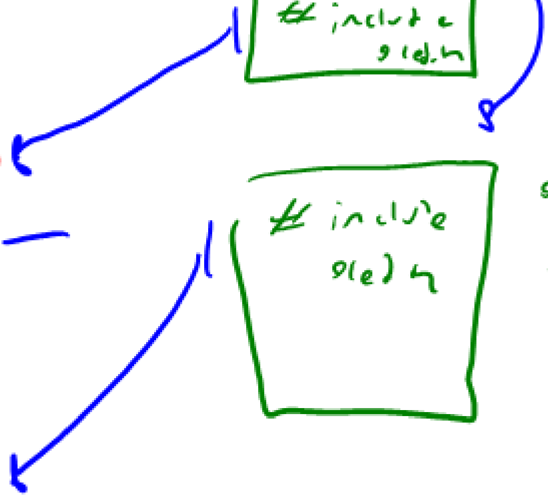
#ifndef OLED_H
#define OLED_H

#endif

#ifndef OLE.h

#endif





oled.h

oled_invert

Main

```
#include oled.h
#include oleddriver

main (
  oled ()
  oled driver()
```

Fundamentals of C

```c
#include <stdio.h>

#define PRODUCT 10

float MultByAddition(float factor, int multiplier);

void main(void){
    float f_new, f_original = 0.0;
    int i = PRODUCT;

    f_new = MultByAddition(f_original, i);

    printf("%f multiplied by %d is %f\n", f_original, i, f_new);

    while (1);
}

float MultByAddition(float factor, int multiplier){
    float product;
    for (; multiplier > 0; multiplier--) {
        product = product + factor;
    }
};
```

Preprocessor dir.

function prototype

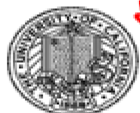
main fcn

// variable declarations

function call

control structure

function definition



Comments

Definition

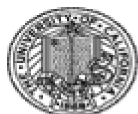
Comments are used to document a program's functionality and to explain what a particular block or line of code does. Comments are ignored by the compiler, so you can type anything you want into them.

- Two kinds of comments may be used:
  - Block Comment

```
/* This is a comment */
```

  - Single Line Comment

```
// This is also a comment
```

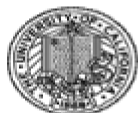


Comments

Using Block Comments

- Block comments:
  - Begin with /* and end with */
  - May span multiple lines

```
/*****************************************************************
* Program: hello.c
* Author:  R. Ostapiuk
*****************************************************************/
#include <stdio.h>

/* Function: main() */
int main(void)
{
  printf("Hello, world!\n"); /* Display "Hello, world!" */
}
```

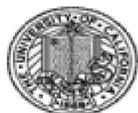


Comments

Using Single Line Comments

- Single line comments:
  - Begin with // and run to the end of the line
  - May *not* span multiple lines

```
//==========================================================
// Program: hello.c
// Author:  R. Ostapiuk
//==========================================================
#include <stdio.h>

// Function: main()
int main(void)
{
  printf("Hello, world!\n"); // Display "Hello, world!"
}
```



Comments

Nesting Comments

- Block comments may not be nested within other delimited comments
- Single line comments may be nested

Example: Single line comment within a delimited comment.

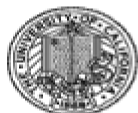
```
/*
    code here       // Comment within a comment
*/
```

Example: Delimited comment within a delimited comment.

Delimiters don't match up as intended!

```
/*
    code here       /* Comment within a comment */
    code here       /* Comment within a… oops!  */
*/
```

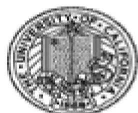
Dangling delimiter causes compile error



Comments

Best Practices/Doxygen

```
/**
 * @file
 * @author R. Ostapiuk
 * @section DESCRIPTION
 * This is an example Hello World program
 */
#include <stdio.h>

/**
 * Main, the entrypoint for this C program.
 * @return A success code, where non-zero values indicate failure
 */
int main(void)
{
    int i;          /// Loop counter variable
    char *p;        /// Pointer to text string

    // Display greeting
    printf("Hello, world!\n");
}
```



Variables and data types

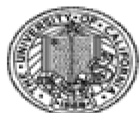
4 basic data types

int

float

double

char



Variables and Data Types

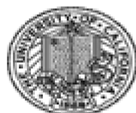
Example

```
#include <stdio.h>

#define PI 3.14159

int main(void)
{
    float radius, area;

    //Calculate area of circle
    radius = 12.0;
    area = PI * radius * radius;
    printf("Area = %f", area);
}
```

Data Types

Variable Declarations

Variables in use



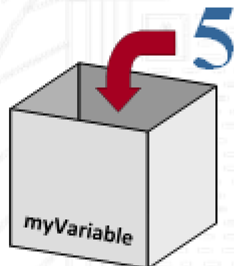
Variables

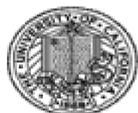
Definition

A variable is a name that represents one or more memory locations used to hold program data.

- A variable may be thought of as a container that can hold data used in a program

```
int myVariable;
myVariable = 5;
```

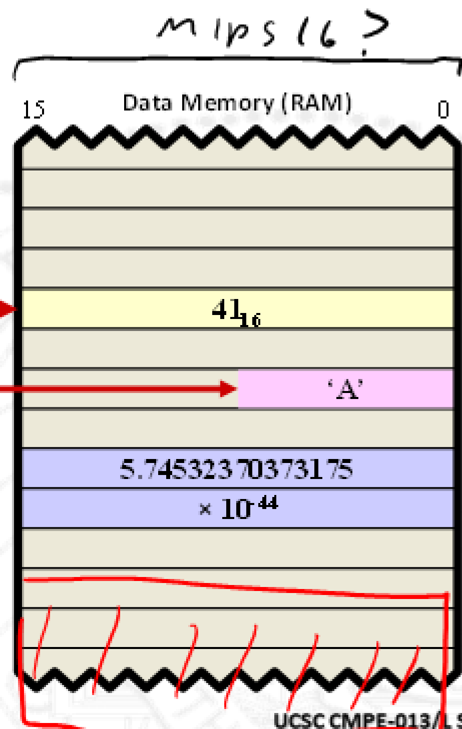


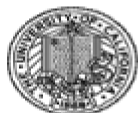


Variables

- Variables are names for storage locations in memory

```
int warp_factor;
char first_letter;
float length;
```

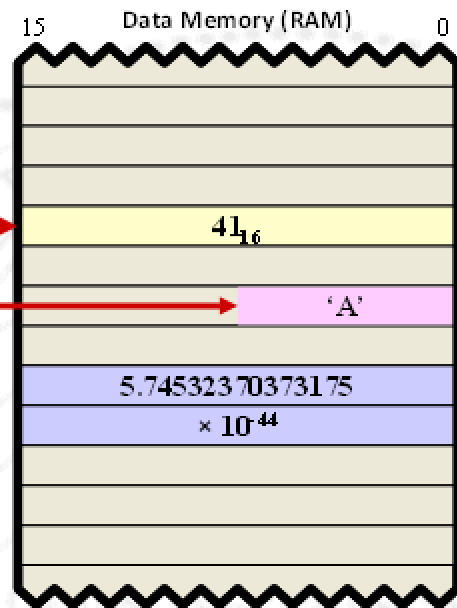


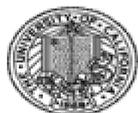


Variables

- Variable declarations consist of a unique identifier (name)…

```
int warp_factor;
char first_letter;
float length;
```

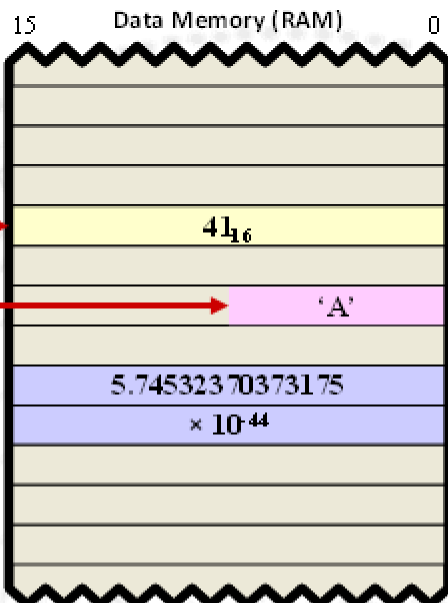


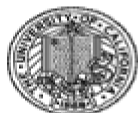


Variables

- ...and a data type
  - Determines size
  - Determines how values are interpreted

```
int warp_factor;
char first_letter;
float length;
```



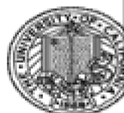


Identifiers

- Names given to program elements:
  - Variables, Functions, Arrays, Other elements

Example of Identifiers in a Program

```
#include <stdio.h>

#define PI 3.14159

int main(void)
{
    float radius, area;

    //Calculate area of circle
    radius = 12.0;
    area = PI * radius * radius;
    printf("Area = %f", area);
}
```



Identifiers

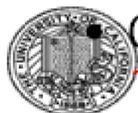
- Valid characters in identifiers:

I	d	e	n	t	i	f	i	e	r

First Character
'_' (underscore)
'A' to 'Z'
'a' to 'z'

Remaining Characters
'_' (underscore)
'A' to 'Z'
'a' to 'z'
'0' to '9'

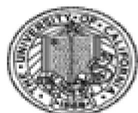
- Case sensitive!
- Only first 31 characters significant*



ANSI C Keywords

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

- Some compiler implementations may define additional keywords



Data Types

Fundamental Types

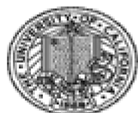
Type	Description	Bits
`char`	single character	8
`int`	integer	16
`float`	single precision floating point number	32
`double`	double precision floating point number	64

The size of an **int** varies from compiler to compiler.

- XC16 **int** as 16-bits
- XC32 defines **int** as 32-bits

If you need precise length variable types, use stdint.h

- uint8_t is unsigned 8 bits
- int16_t is signed 16bits, etc.

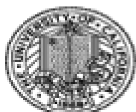


Data Type Qualifiers

Modified Integer Types

Qualifiers: unsigned, signed, short and long

Qualified Type	Min	Max	Bits
unsigned char	0	255	8
char, signed char	-128	127	8
unsigned short *int*	0	65535	16
short *int*, signed short *int*	-32768	32767	16
unsigned *int*	0	65535	16
int, signed *int*	-32768	32767	16
unsigned long *int*	0	$2^{32}-1$	32
long *int*, signed long *int*	-2^{31}	$2^{31}-1$	32
unsigned long long *int*	0	$2^{64}-1$	64
long long *int*, signed long long *int*	-2^{63}	$2^{63}-1$	64



Short

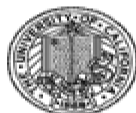
char ≤ Size of (short int) ≤ size of (int) ≤ size of (long int)

Data Type Qualifiers

Modified Floating Point Types

Qualified Type	Absolute Min	Absolute Max	Bits
`float`	$\pm\sim10^{-44.85}$	$\pm\sim10^{38.53}$	32
`double`[1]	$\pm\sim10^{-44.85}$	$\pm\sim10^{38.53}$	32
`long double`	$\pm\sim10^{-323.3}$	$\pm\sim10^{308.3}$	64

MPLAB-X XC32 Uses the IEEE-754 Floating Point Format



Variables

How to Declare a Variable

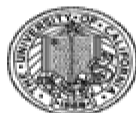
Syntax

```
type identifier1, identifier2,…,identifiern;
```

- A variable must be declared before it can be used
- The compiler needs to know how much space to allocate and how the values should be handled

Example

```
int x, y, z;
float warpFactor;
char text_buffer[10];
unsigned index;
```



Variables

How to Declare a Variable

Variables may be declared in a few ways:

Syntax

One declaration on a line

```
type identifier;
```

One declaration on a line with an initial value

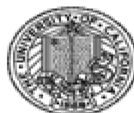
```
type identifier = InitialValue;
```

Multiple declarations of the same type on a line

$type\ identifier_1,\ identifier_2,\ identifier_3;$

Multiple declarations of the same type on a line with initial values

$type\ identifier_1 = Value_1,\ identifier_2 = Value_2;$



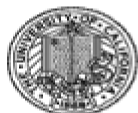
Variables

How to Declare a Variable

Examples

```
unsigned int x;
unsigned y = 12;
int a, b, c;
long int myVar = 0x12345678;
long z;
char first = 'a', second, third = 'c';
float big_number = 6.02e+23;
```

It is customary for variable names to be spelled using "camel case", where the initial letter is lower case. If the name is made up of multiple words, all words after the first will start with an upper case letter (e.g. `myLongVarName`).



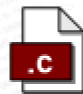
Variables

How to Declare a Variable

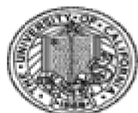
- Sometimes, variables (and other program elements) are declared in a separate file called a header file
- Header file names customarily end in .h

- Header files are associated with a program through the `#include` directive

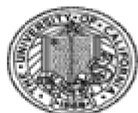
MyProgram.h



MyProgram.c

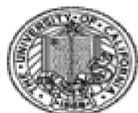


Control Structures



Expressions

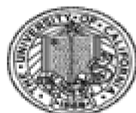
- Represents a single data item (e.g. character, number, etc.)
- May consist of:
  - A single entity (a constant, variable, etc.)
  - A combination of entities connected by operators (+, -, * , / and so on)



Expressions

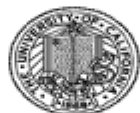
Example

```
a + b;
x = y;  ← assignment
speed = dist / time;
z = ReadInput();
c <= 7;        Is True (c<=7)
x == 25;
count++;
d = a + 5;
```



Statements

- Cause an action to be carried out
- Three kinds of statements in C:
  - Expression Statements
  - Compound Statements
  - Control Statements

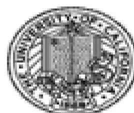


Expression Statements

- An expression followed by a semi-colon
- Execution of the statement causes the expression to be evaluated

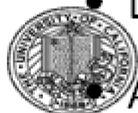
Examples

```
i = 0;
i++;
a = 5 + i;
y = (m * x) + b;
printf("Slope = %f", m);
;
```



Compound Statements

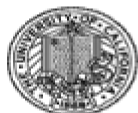
- A group of individual statements enclosed within a pair of curly braces { and }
- Individual statements within may be any statement type, including compound
- Allows statements to be embedded within other statements
- Does NOT end with a semicolon after }
- Also called Block Statements



Compound Statements

Example

```
{
  float start, finish;

  start = 0.0;
  finish = 400.0;
  distance = finish - start;
  time = 55.2;
  speed = distance / time;
  printf("Speed = %f m/s", speed);
}
```



```
for (x = 0; x < 10; x++) {
  print("...");
  print( ~~~
  ~~~~~~
}
```

```
{
  int i;
  // do stuff w/ i
}
```

/* ~~~~ */

//

```
for (x = 0; x < 10; {x++, y++}) {
    y++
```

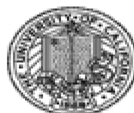
Control Statements

- Used for loops, branches and logical tests
- Often require other statements embedded within them

Example

```
while (distance < 400.0) {
    printf("Keep running!");
    distance += 0.1;
}
```

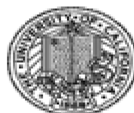
(while syntax: `while` *expr statement*)



Boolean Expressions

- Boolean data type added in C99
- Boolean expressions return integers:
  - 0 expressions evaluate as false
  - non-zero expressions evaluate as true (generally 1)

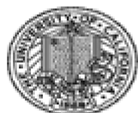
```
{
    int x = 5;
    bool y, z;

    y = (x > 4);    // y = true (1)
    z = (x > 6);    // z = false (0)
    while (1);
}
```



Boolean Expressions

Equivalent Expressions

- If a variable, constant, or function call is used alone as the conditional expression:
  `(myVar)` or `(Foo())`
- This is the same as saying:
  `(myVar != 0)` or `(Foo() != 0)`
- In either case, if `myVar` $\neq 0$ or `Foo()` $\neq 0$, then the expression evaluates as true (non-zero)



`if` Statement

Syntax

```
if (expression) statement
```

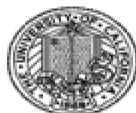
- *expression* is evaluated for boolean true(≠0) or false (=0)
- If true, then *statement* is executed

Note

Whenever you see ***statement*** in a syntax guide, it may be replaced by a compound (block) statement.

Remember: spaces and new lines are not significant.

```
if (expression) {
    statement1
    statement2
}
```

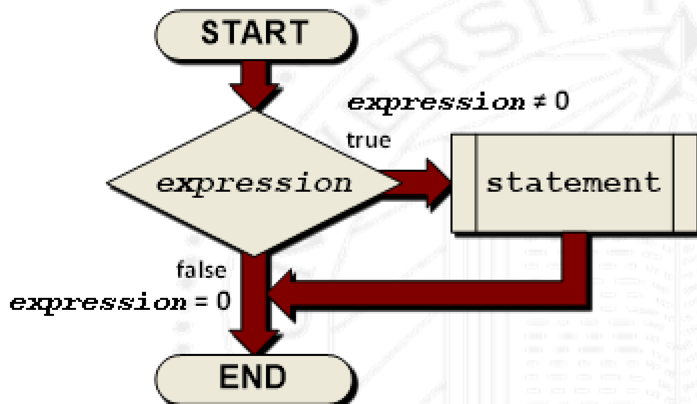


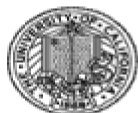
`if` Statement

Syntax

Flow Diagram

```
if (expression) statement
```



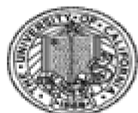


if Statement

Example

```
{
   int x = 5;

   if (x) {                          If x is TRUE (non-zero)...
      printf("x = %d\n", x);        ...then print the value of x.
   }
   while (1);
}
```

- What will print if x = 5? ... if x = 0?
- ...if x = -82?
- ...if x = 4294967296?



`if` Statement

Testing for TRUE

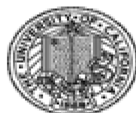
- `if (x)` vs. `if (x == 1)`
  - `if (x)` only needs to test for not equal to 0
  - `if (x == 1)` needs to test for equality with 1
  - Remember: true is defined as non-zero, false is defined as zero

Example: if (x)

```
if (x)

8:                      if (x)
011B4   E208C2          cp0.w 0x08c2
011B6   320004          bra z, 0x0011c0
```

Example: if (x ==1)

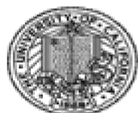
```
if (x == 1)

11:                     if (x == 1)
011C0   804610          mov.w 0x08c2,0x0000
011C2   500FE1          sub.w 0x0000,#1,[0x001e]
011C4   3A0004          bra nz, 0x0011ce
```



Nested `if` Statements

Example

```
int power = 10;
float band = 2.0;
float frequency = 146.52;

if (power > 5) {
   if (band == 2.0) {
      if ((frequency > 144) && (frequency < 148)) {
        printf("Yes, it's all true!\n");
      }
   }
}
```

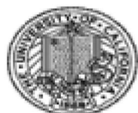


`if-else` Statement

Syntax

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

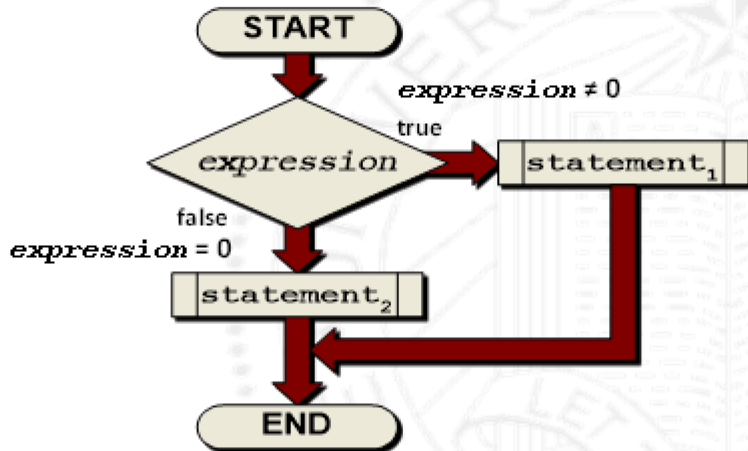
- *expression* is evaluated for boolean true(≠0) or false (=0)
- If true, then $statement_1$ is executed
- If false, then $statement_2$ is executed

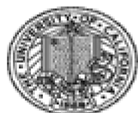


`if-else` Statement

Syntax

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

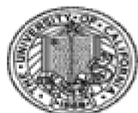




`if-else` Statement

Example

```
float frequency = 146.52;  // Frequency in MHz

if ((frequency > 144.0) && (frequency < 148.0)) {
    printf("You're on the 2 meter band\n");
} else {
    printf("You're not on the 2 meter band\n");
}
```

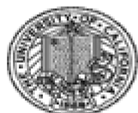


`if-else if` Statement

Syntax

```
if (expression1) statement1
else if (expression2) statement2
else statement3
```

- $expression_1$ is evaluated for boolean true (≠0) or false (=0)
- If true, then $statement_1$ is executed
- If false, then $expression_2$ is evaluated
- If true, then $statement_2$ is executed
- If false, then $statement_3$ is executed



```
if (X) {
  //do a
}
if (Y) {
  //do b
}
```

vs

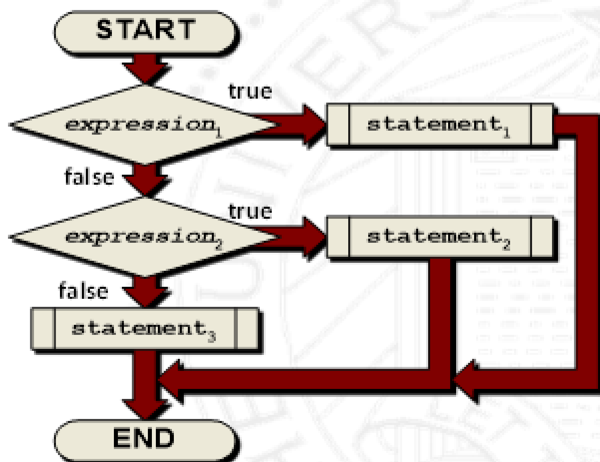
```
if (X) {
  //do a
} else if (Y) {
  //do b
}
```

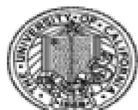
X	Y		
0	0	— —	— —
0	1	— b	— b
1	0	a —	a —
1	1	a b	a —

`if-else if` Statement

Syntax

```
if (expression1) statement1
else if (expression2) statement2
else statement3
```

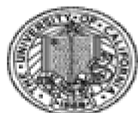




if-else if Statement

Example

```
if ((freq > 144) && (freq < 148)) {
    printf("You're on the 2 meter band\n");
} else if ((freq > 222) && (freq < 225)) {
    printf("You're on the 1.25 meter band\n");
} else if ((freq > 420) && (freq < 450)) {
    printf("You're on the 70 centimeter band\n");
} else {
    printf("You're somewhere else\n");
}
```

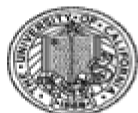


while Loop

Syntax

```
while (expression) statement
```

- If *expression* is true, *statement* will be executed and then *expression* will be re-evaluated to determine whether or not to execute *statement* again
- It is possible that *statement* will never execute if *expression* is false when it is first evaluated



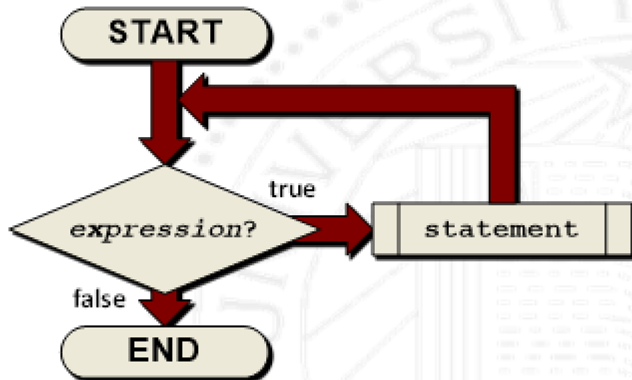
`while` Loop

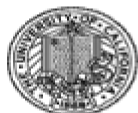
Syntax

Flow Diagram

```
while (expression) statement
```

while (x) {
}





```
         init    cond   operation
          ↓       ↓       ↓
for ( x ; y ; z ) {
    // stuff
}
```

```
x;
while (y) {
    // stuff
    z;
}
```

```
while(Q){
  //do stuff
}
```

⇒

```
for(;Q;){

}
```

```
while(1);
```

⇒

```
for(;;)
```

```
x = 0;
do {
  // stuff
} while( x ...
```

```
x = 0;
while(x) {
  // stuff
}
```

```
x = 0

do {

  // do stuff

} while(x)
```

=>

```
x = 0;

while (1) {

  // do stuff

  if (x == 0) {
    break; }
}
```

```
while (1) {

    //

    for (        ) {
        if sensor (i) is tripped:
            break;
    }

}
```

~~for i = 2 to N~~

~~for j = 2 to i~~

does ~~i / j = int~~ ?

while Loop

Example (Code Fragment)

```
int i = 0;

while (i < 5) {
        printf("Loop iteration %d\n", i++);
}
```

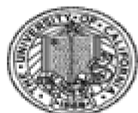
Loop counter initialized outside of loop

Condition checked at start of loop iterations

Loop counter incremented manually inside loop

Expected Output:

```
Loop iteration 0
Loop iteration 1
Loop iteration 2
Loop iteration 3
Loop iteration 4
```



while Loop

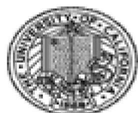
- Primary looping mechanism
- Completely generic
- Frequently used for main loop of program

Generic loop:

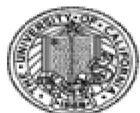
```
while (HaveData()) {
    PrintData();
}
```

Main loop:

```
while (1) {
    ...
}
```



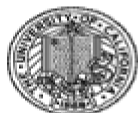
Functions



Functions

Program Structure

```
main()
{
   ...
   eat();
   ...
   drink();
   ...
}

eat()
{
   ...
   return;
}

drink()
{
   ...

   be_merry();
   return;
}

be_merry()
{
   ...
   return;
}
```



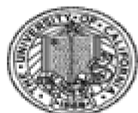
Functions

What is a function?

Definition

Functions are self contained program segments designed to perform a specific, well defined task.

- All C programs have one or more functions
- The `main()` function is required
- Functions can accept parameters from the code that calls them
- Functions return a single value (but can export more data)
- Functions help to organize a program into logical, manageable segments



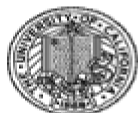
Functions

Remember Algebra Class?

- Functions in C are conceptually like an algebraic function from math class…

	Function Definition
Function Name →	$f(x) = x^2 + 4x + 3$
	↑ **Function Parameter**

- If you pass a value of 7 to the function: f(7), the value 7 gets "copied" into *x* and used everywhere that *x* exists within the function definition: $f(7) = 7^2 + 4*7 + 3 = 80$



Functions

Definitions

Syntax

Data type of return *expression*

Name

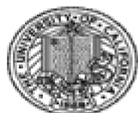
Parameter List (optional)

```
type identifier(type1 arg1,…,typen argn)
{
    declarations
    statements
    return expression;
}
```

Body

Header

Return Value (optional)



Functions

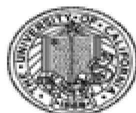
Function Definitions: Syntax Examples

Example

```c
int Maximum(int x, int y)
{
  int z;

  z = (x >= y) ? x : y;
  return z;
}
```

Example – A more efficient version

```c
int Maximum(int x, int y)
{
  return ((x >= y) ? x : y);
}
```



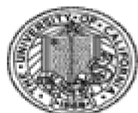
Functions

Function Definitions: Return Data Type

Syntax

```
type identifier(type1 arg1,…,typen argn)
{
  declarations
  statements
  return expression;
}
```

- A function's ***type*** must match the type of data in the return ***expression***



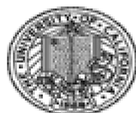
Functions

Function Definitions: Return Data Type

- A function may have multiple return statements, but only one will be executed and they must all be of the same type

Example

```
int bigger(int a, int b)
{
    if (a > b) {
        return 1;
    } else {
        return 0;
    }
}
```



Functions

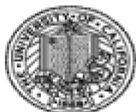
Function Definitions: Return Data Type

- The function type is `void` if:
  - The `return` statement has no *expression*
  - The `return` statement is not present at all
- This is sometimes called a *procedure function* since nothing is returned

Example

```
void identifier(type1 arg1,…,typen argn)
{
  declarations
  statements
  return;
}
```

`return;` may be omitted if nothing is being returned



Functions

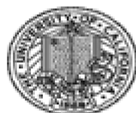
Function Definitions: Parameters

- A function's parameters are declared just like ordinary variables, but in a comma delimited list inside the parentheses
- The parameter names are only valid inside the function (local to the function)

Syntax

```
type identifier(type1 arg1,…,typen argn)
{
  declarations
  statements
  return expression;
}
```

Function Parameters



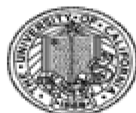
Functions

Function Definitions: Parameters

- Parameter list may mix data types
  - `int Foo(int x, float y, char z)`
- Parameters of the same type must be declared separately – in other words:
  - `int Maximum(int x, y)` will not work
  - `int Maximum(int x, int y)` is correct

Example

```
int Maximum(int x, int y)
{
  return ((x >= y) ? x : y);
}
```



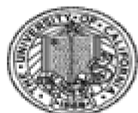
Functions

Function Definitions: Parameters

- If no parameters are required, use the keyword `void` in place of the parameter list when defining the function

Example

```
type identifier(void)
{
        declarations
        statements
        return expression;
}
```



Functions

How to Call / Invoke a Function

Function Call Syntax

- No parameters and no return value

```
Foo();
```

- No parameters, but with a return value

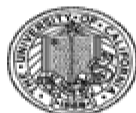
```
x = Foo();
```

- With parameters, but no return value

```
Foo(a, b);
```

- With parameters and a return value

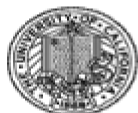
```
x = Foo(a, b);
```



Functions

Function Prototypes

- Just like variables, a function must be declared before it may be used
- Declaration must occur before main() or other functions that use it
- Declaration may take two forms:
  - The entire function definition
  - Just a function prototype – the function definition itself may then be placed anywhere in the program



Functions

Function Prototypes

- Function prototypes may be take on two different formats:
  - An exact copy of the function header:

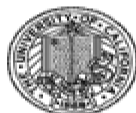
Example – Function Prototype 1

```
int Maximum(int x, int y);
```

  - Like the function header, but without the parameter names – only the types need be present for each parameter (bad form!):

Example – Function Prototype 2

```
int Maximum(int, int);
```

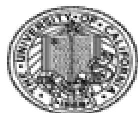


Functions

Example 1

```
int a = 5, b = 10, c;

int Maximum(int x, int y)
{
   return ((x >= y) ? x : y);
}

int main(void)
{
   c = Maximum(a, b);
   printf("The max is %d\n", c)
}
```

Function is *declared* and *defined* before it is used in main()



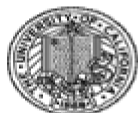
Functions

Example 2

```
int a = 5, b = 10, c;

int Maximum(int x, int y);

int main(void)
{
  c = Maximum(a, b);
  printf("The max is %d\n", c)
}

int Maximum(int x, int y)
{
  return ((x >= y) ? x : y);
}
```

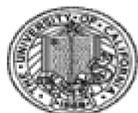
Function is *declared* with prototype before use in main()

Function is *defined* after it is used in main()



Roadmap

- Announcements, lab kits
- Git, theory and practice
- Fundamental Elements of C
  - Comments
  - Variables + datatypes
  - Preprocessor macros
  - Control structures (?)
- MPLAB X tour



Questions?