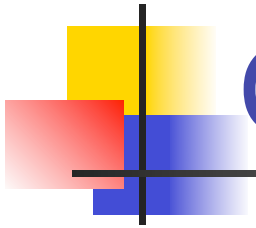
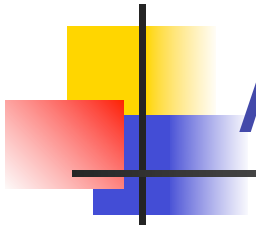


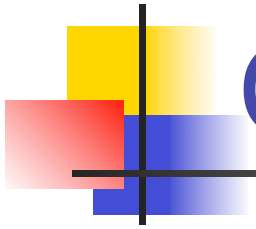
CS 11 C track: lecture 1

- Preliminaries
  - Need a CS (CMS) cluster account
    - http://acctreq.cms.caltech.edu/cgi-bin/request.cgi
  - Need to know UNIX
    - ITS tutorial linked from track home page
  - Track home page:
    - http://courses.cms.caltech.edu/cs11/material/c/mike/index.html



Assignments

- 1st assignment is posted now
- Due one week after class, midnight
- Grading policy: see "admin page" linked from track home page

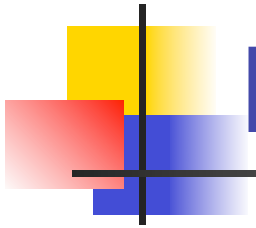


Other administrative stuff

- Again, see admin web page:

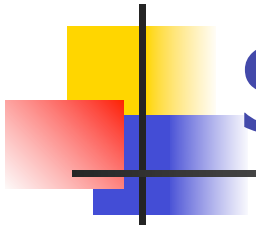
http://courses.cms.caltech.edu/cs11/material/c/mike/admin.html

- Covers how to submit labs, collaboration policy, grading, etc.



Main textbook

- C: A Software Engineering Approach, 3rd ed.
  by Peter A. Darnell, Philip E. Margolis
- Thorough, readable

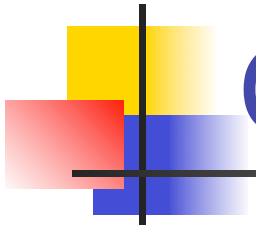


Supplemental textbook

- Kernighan and Ritchie:

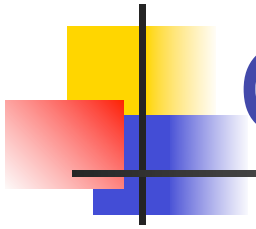
The C Programming Language, 2nd. ed.

  - 1st edition NOT acceptable
  - "ANSI C"
  - Good for reference



C: pros and cons

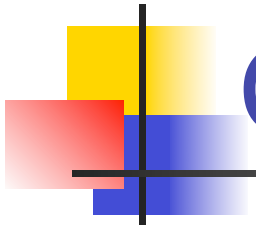
- What C is good at
  - low-level programming
  - speed and memory efficiency
  - portability (sorta)
- Bad things about C
  - unsafe!!!
  - low level of abstraction



Getting started (1)

- The "hello, world!" program:

```
#include <stdio.h>
int main(void)
{
    printf("hello, world!\n");
    return 0;
}
```

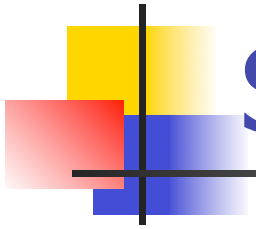


Getting started (2)

- Make this into a file called hello.c using a text editor
  - e.g. emacs, vi, nedit, pico
- Compile into a program and run:

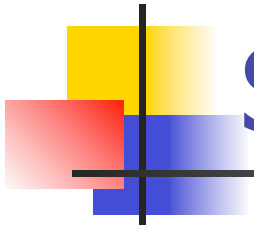
```
% gcc hello.c -o hello
% hello
hello, world!
%
```

- Woo hoo!



Source code to executable (1)

- What you write is called "source code"
- Two kinds of source code files:
  - regular code (files end in ".c")
  - header files (files end in ".h")
- Compiler turns source code into "object code"
  - (files end in ".o")
- Linker turns object code file(s) into executable (no special file suffix)

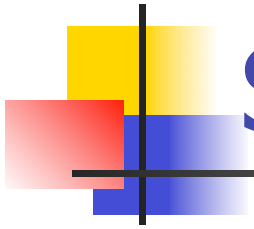


Source code to executable (2)

- The program gcc is both a compiler and a linker
- When you do this:

```
% gcc hello.c -o hello
```

- Then gcc
  - compiles hello.c to hello.o
  - links hello.o with system libraries
  - outputs the binary executable program hello
  - removes hello.o

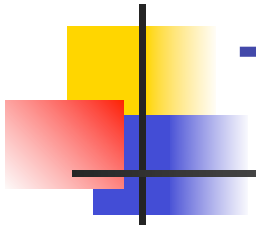


Source code to executable (3)

- You can do each step individually:

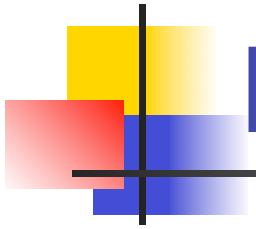
```
% gcc -c hello.c          (compile only)
% gcc hello.o -o hello    (link only)
```

- In this case, hello.o is not removed
- Sequence:
  - compiling: source code to object code
  - linking: object code to binary executable



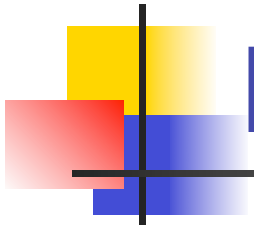
The C language - overview

- Programs are built up of functions
- Functions
  - take in arguments
  - compute something
  - return a result
- The `main()` function
  - is where program execution starts



Data types (1)

- All data in C has to have a specified type
- Examples:
  - `int` (integer)
  - `char` (character)
  - `float` or `double` (approximate real number)
  - others
- Variables hold data of a particular type only
- Variables must be declared before use

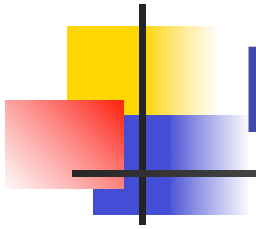


Data types (2)

- Type declarations:

```
int i;    /* name = i   type = int */
char c;   /* name = c   type = char */
double d;
float some_float = 3.14;
```

- Identifiers: `i`, `c`, `d`, `some_float`
- Optional initialization (e.g. `some_float`)
- Booleans → `0` or nonzero (usually `1`)

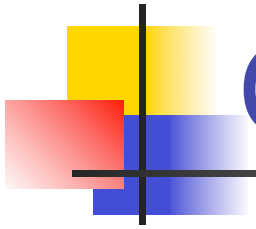


Data types (3)

- Strings: arrays of type **`char`**

```
char some_string[9] = "woo hoo!";

char same_string[] = "woo hoo!";
```

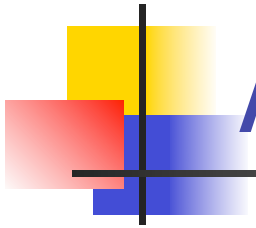
- Much more on strings, arrays later
- Other types: structs, pointers



Operators (1)

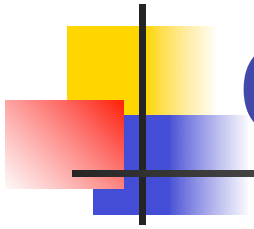
- Numeric: `+ - * / %`
- Assignment: `=`

```
int i = 10;     /* initialization */
int j = 20;     /* initialization */
i = 2 + i * j; /* assignment  */
j = j % 2;      /* assignment  */
```



Assignment operator

- Assignment works this way:
  - 1) Evaluate the right-hand side (RHS) of the assignment operator
  - 2) Assign the resulting value to the left-hand-side (LHS) of the assignment operator



Operators (2)

- What does

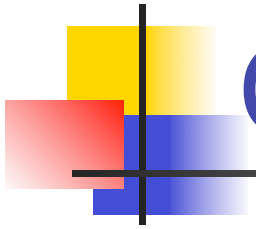
```
i = 2 + i * j;
```

mean?

a) `i = (2 + i) * j;`

b) `i = 2 + (i * j);`

- `*` has a higher precedence than `+`
- Use `()` to force other interpretation



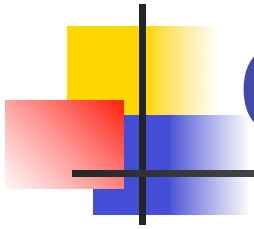
Operators (3)

- Other assignment operators:
  - `+=, -=, *=, ...`

```
i += 2;   /* i = i + 2; */
```

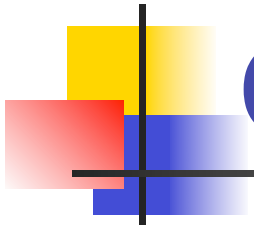
  - increment and decrement: `++, --`

```
i++;      /* i = i + 1; */

++i;      /* same */
```



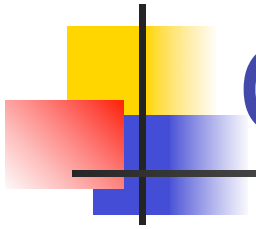
Operators (4)

- Test operators:
  - compare two values
  - < <= > >=
  - == for testing equality
  - != for testing inequality
  - read "!" as "not"



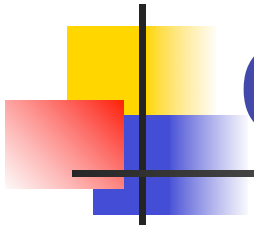
Operators (5)

- Logical operators:
  - arguments are `ints` used as booleans
  - i.e. usually `0` or `1` (false or true)
  - `!` operator is unary logical "not"
  - `&&` operator is binary logical "and"
  - `||` operator is binary logical "or"



Operators (6)

```
int bool1, bool2, bool3, bool4;

bool1 = 0;  /* false */

bool2 = !bool1;  /* bool2 --> true */

bool3 = bool1 || bool2;  /* value? */

bool4 = bool1 && bool2;  /* value? */
```

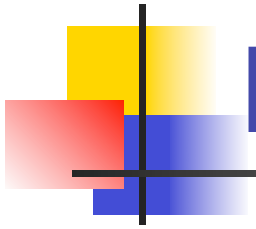


Operators (7)

- "Unary minus" operator:

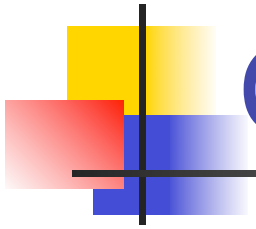
```
int var1 = 10;

int var2;

var2 = -var1;
```

- Like – with nothing to the left
- Negates the value



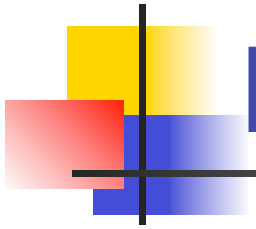
Expressions and statements

- `i + 2 * j` is an expression (has a value)
- `i = j * k;` is a statement
  - ends in a semicolon
  - also is an expression (value is value of `i`)
- `i = j = k = 0;` is allowed
- Equivalent to `i = (j = (k = 0));`
- NOT `((i = j) = k) = 0;`



Comments

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

/*

 * Comments can span

 * multiple lines.

 */

// This is NOT a comment!
```



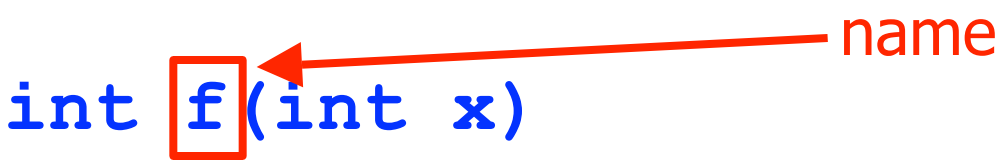
Functions (1)

- Functions take arguments and return values:

```
int f(int x)
{
    int y = 10;
    return y * x;
}
```

Functions (2)

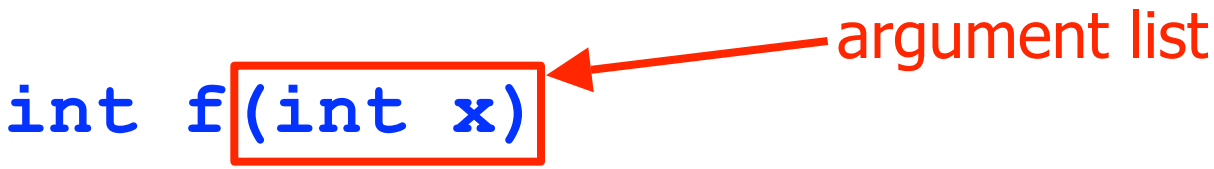
- Functions take arguments and return values:




```
int f(int x)
{
    int y = 10;
    return y * x;
}
```

Functions (3)

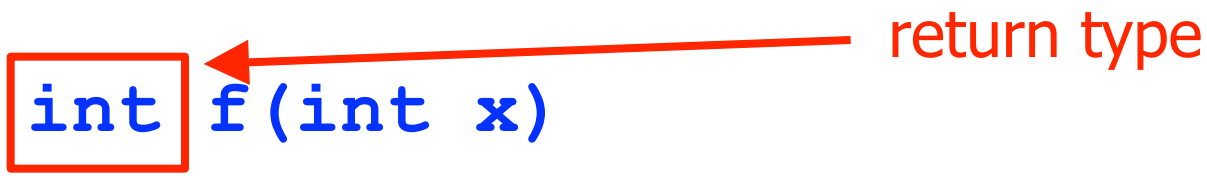
- Functions take arguments and return values:



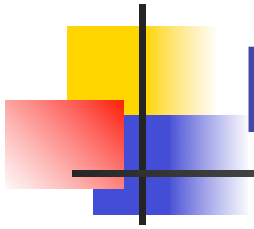

```
int f(int x)
{
    int y = 10;
    return y * x;
}
```

Functions (4)

- Functions take arguments and return values:

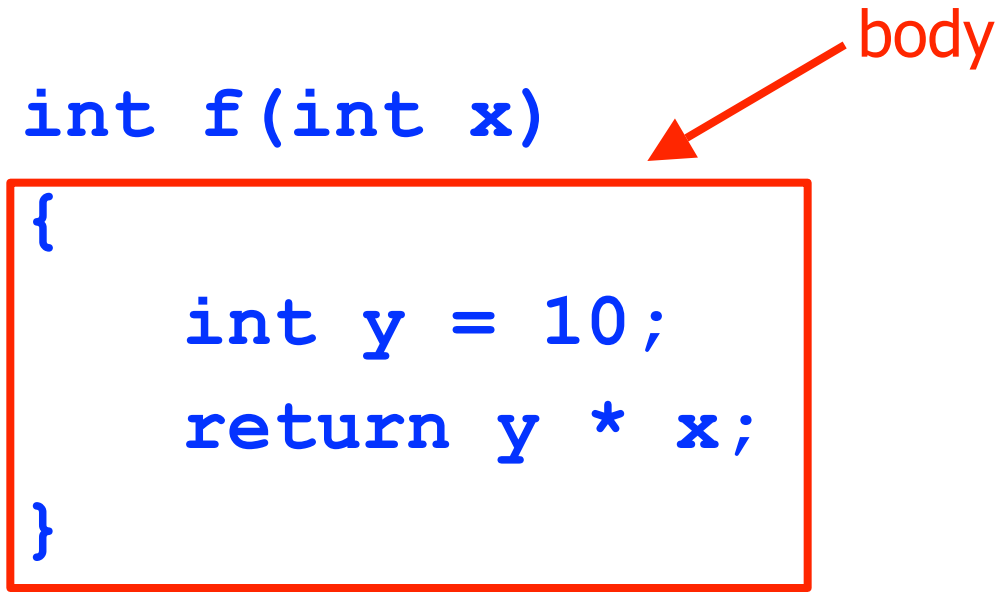



```
int f(int x)
{
    int y = 10;
    return y * x;
}
```



Functions (5)

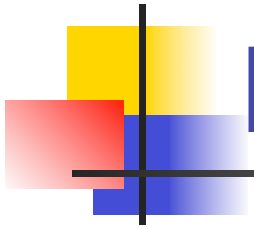
- Functions take arguments and return values:



Functions (6)

- Functions take arguments and return values:

```
int f(int x)
{
    int y = 10;
    return y * x;
}
```



Functions (7)

- Calling the function we just defined:

```
/* in another function... */
int res;
int i = 10;
res = f(10);
res = f(5 + 5);
res = f(i);
res = f(i*5 + i/2);
```

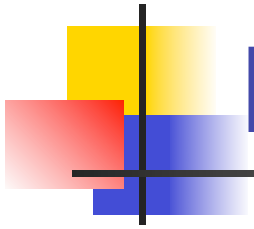
- All of these are valid function calls
- Take in arguments, return result

Functions (8)

- Functions can take multiple arguments:


```
int g(int x, int y)
{
    int z = 42;
    return x * y * z;
}
```

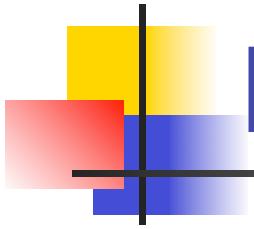
- Argument names (`x`, `y`) preceded by type (`int`)
- Arguments separated by commas



Functions (9)

- Calling functions that take multiple arguments:

```
/* in another function... */
int res;
int i = 10, j = 20;
res = g(10, 20);
res = g(5 + 5, 20);
res = g(i, j);
res = g(i*5 + i/2, j * 10);
```

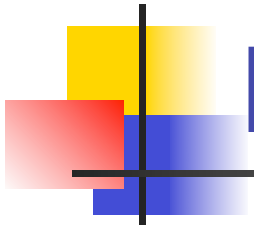


Functions (10)

- Not all functions return values:

```
void print_number(int i)
{
    printf("number is: %d\n", i);
}
```

- Return type is void (nothing to return)
- Use this when no return value needed

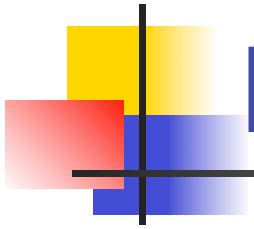


Functions (11)

- Not all functions return values:

```
void print_number(int i)
{
    printf("number is: %d\n", i);
    return; /* unnecessary */
}
```

- return statement not required
  - unless you return in the middle of the function

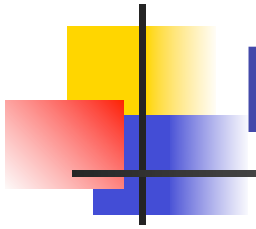


Functions (12)

- Calling this function:

```
/* In another function... */
int i = 10;
print_number(20);
print_number(i);
print_number(i*5 + i/2);
```

- Prints `20`, `10`, `55` respectively

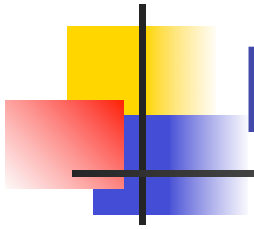


Functions (13)

- Not all functions take arguments:

```
int five(void)
{
    return 5;
}
```

- No arguments (use void to indicate)



Functions (14)

- Calling functions without arguments:

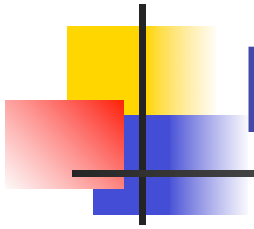
```
int value;
value = five();
```

- Now value equals `5`
- Note `()` after `five`
  - means "this function is being called with no arguments"
  - Without this, function won't be called!

Functions – type declarations

- Type declarations, if needed, come at the beginning of the function
- Need a declaration for every local variable

```
int foo(int x)
{
    int y;  /* type declaration */
    y = x * 2;
    return y;
}
```

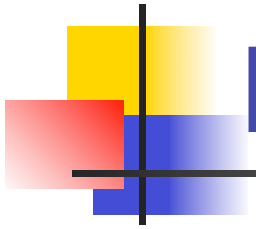


Functions – type declarations

- This is wrong:

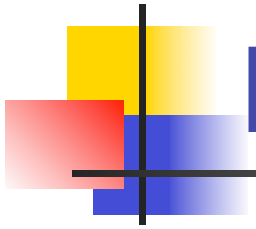
```
int foo(int x)
{
    int y;          /* type decl */
    y = x * 2;      /* code */
    /* type declaration after code: */
    int z = y * y;
    return z;
}
```

- Generates a compiler warning



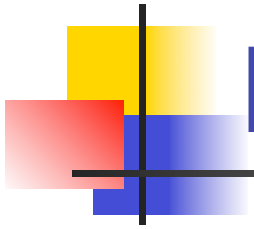
Local and global variables (1)

- Variable declarations can be local or global
- Local: inside a function
- Global: outside a function
  - accessible from any function



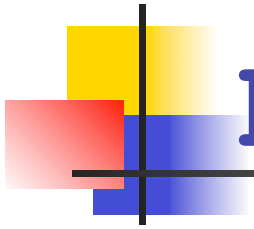
Local and global variables (2)

```
int x;          /* Global variable */
int y = 10;     /* Initialized global variable */


int foo(int z)
{
  int w;    /* local variable */
  x = 42;   /* assign to a global variable */
  w = 10;   /* assign to a local variable */
  return (x + y + z + w);
}
```



Local and global variables (3)

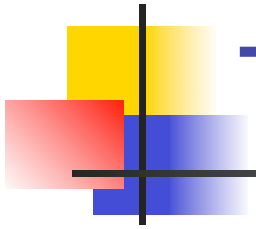
- In general, avoid using global variables!
- Global variables can be changed by any function
  - makes debugging much harder
- Global variables are never necessary
  - though sometimes convenient
- OK to use global "variables" if they really are constant
  - i.e. if you don't change their values



printf()

```
int a = 5;
double pi = 3.14159;
char s[] = "I am a string!";
printf("a = %d, pi = %f, s = %s\n",
        a, pi, s);
```

- Substitutes values for `%d`, `%f`, `%s` etc.
- `%d` : int, `%f` : float, double, `%s` : string
- `\n` : new line

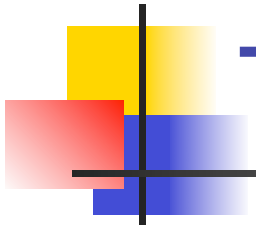


The C preprocessor (1)

- What does the funky line

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

  mean?
- C preprocessor directive
- Extra step in compilation:
  - cpp: source code -> expanded source code
  - gcc: compiles source code -> object code
  - gcc (ld): links object code -> executable
  - gcc does all this for you

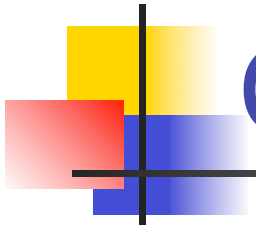


The C preprocessor (2)

- What does the funky line

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

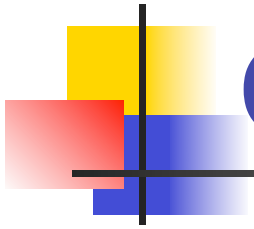
  mean?
- Includes the declaration of `printf()`
  - NOT the implementation
  - allows your code to use `printf()`
- The linker adds in the implementation



Conditionals (1)

- Need to be able to test for conditions:

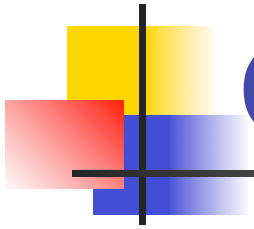
```
int a = 10;
if (a < 20)
{
    printf("less than 20\n");
}
else
{
    printf("not less than 20\n");
}
```



Conditionals (2)

- Test: `0` is “false”, anything else is “true”:

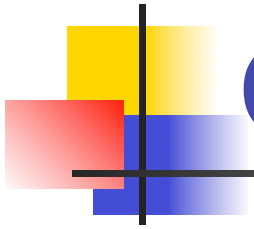
```
if (1) /* true */
{
    printf("less than 20\n");
}
else
{
    printf("not less than 20\n");
}
```



Conditionals (3)

- VERY common error:

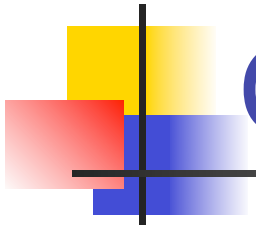
```
int a = 0;
if (a = 10) /* always true! */
{
    printf("a equals 10\n");
}
else
{
    printf("a doesn't equal 10\n");
}
```



Conditionals (4)

- Should be:

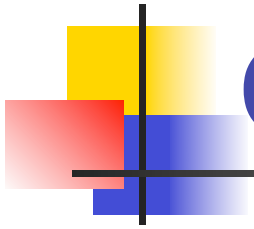
```
int a = 0;
if (a == 10) /* not always true */
{
    printf("a equals 10\n");
}
else
{
    printf("a doesn't equal 10\n");
}
```



Conditionals (5)

- **else** clause is optional:

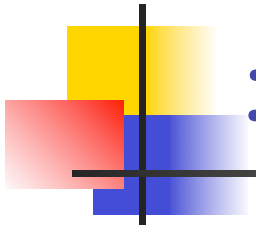
```
int a = 0;
if (a == 10)
{
    printf("a equals 10\n");
}
```



Conditionals (5)

- **`else if`** for multiple cases:

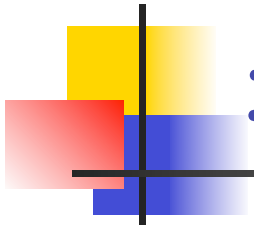
```
int a = 0;
if (a == 10) {
    printf("a equals 10\n");
} else if (a < 10) {
    printf("a is less than 10\n");
} else {
    printf("a is greater than 10\n");
}
```



`for` loop (1)

- Need to do things repeatedly:

```
int i;
for (i = 0; i < 10; i++)
{
    printf("cowabunga!!!\n");
}
```



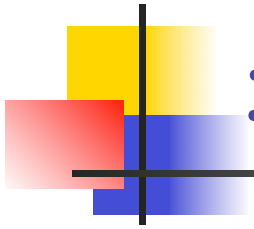
for loop (2)

```
for (<initialization>;
      <test>;
      <increment>)
{ <body> }

for (i = 0; i < 10; i++)
{
    printf("cowabunga!!!\n");
}
```

for loop (3)

```
for (<initialization>;
      <test>;
      <increment>)
{ <body> }
```

```
for (i = 0; i < 10; i++)
{
    printf("cowabunga!!!\n");
}
```



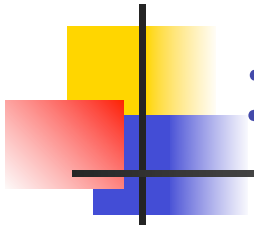
for loop (4)

```
for (<initialization>;
      <test>;
      <increment>)
{ <body> }

for (i = 0; i < 10; i++)
{
    printf("cowabunga!!!\n");
}
```

for loop (5)

```
for (<initialization>;
      <test>;
      <increment>)
{ <body> }
```

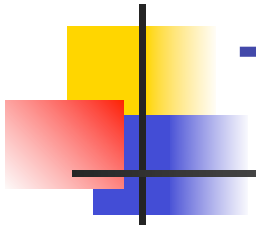
```
for (i = 0; i < 10; i++)
{
    printf("cowabunga!!!\n");
}
```



for loop (6)

```
for (<initialization>;
     <test>;
     <increment>)
{ <body> }
```

```
for (i = 0; i < 10; i++)
{
    printf("cowabunga!!!\n");
}
```



That’s all for now!

- Much more on all these topics in later lectures
- Do first assignment to get familiar with basics
- Use "style checker" to avoid style mistakes
- Have fun!