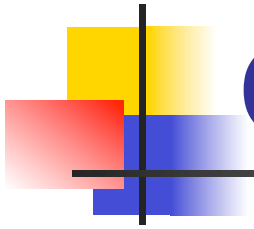
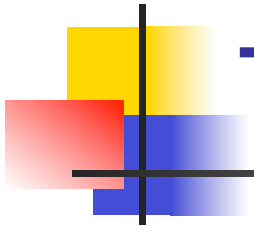


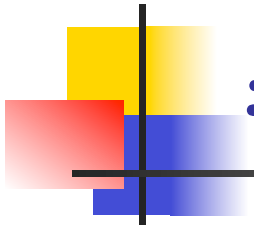
CS 11 C track: lecture 2

- Last week: basics of C programming
  - compilation
  - data types (`int`, `float`, `double`, `char`, etc.)
  - operators (`+ - * / = == +=` etc.)
  - functions
  - conditionals
  - loops
  - preprocessor (`#include`)



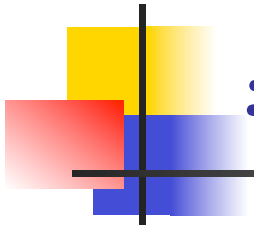
This week

- Preprocessor (`#define`)
- Operators and precedence
- Types and type conversions
- Function prototypes
- Loops (`while, do/while`)
- More on input/output and `scanf()`
- Commenting
- Using the `make` program



`#define` (1)

- So far, only preprocessor command we know is **`#include`**
- Lots of other ones as well
  - will see more later in course
- One major one: **`#define`**
- Used in almost all C header files

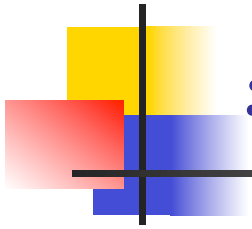


`#define` (2)

- `#define` usually used to define symbolic constants:

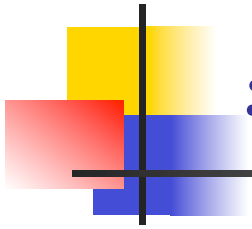
```
#define MAX_LENGTH 100
```

- Then preprocessor substitutes the number `100` for `MAX_LENGTH` everywhere in program
- NOTE: Just a textual substitution!
  - no type checking



#define (3)

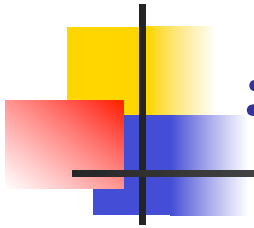
```
#define MAX_LENGTH 100
/* later... */
int i;
/* later... */
if (i > MAX_LENGTH) {
    printf("Whoa there!\n");
}
```



#define (4)

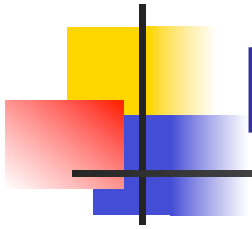
```
/* That code expands into: */
if (i > 100) {
    printf("Whoa there!\n");
}
```

- Note that all occurrences of **`MAX_LENGTH`** replaced with `100`
- Why not just write `100` in the first place?



#define (5)

- Why not just write `100` in the first place?
- If you decide you want to change `MAX_LENGTH` to another number instead
  - only have to change one `#define` statement and all occurrences of `MAX_LENGTH` will be changed to the new number
- Hard-coded numbers like `100` are called magic numbers
  - usually repeated many times in a program
  - would have to change many lines to change the number throughout the program

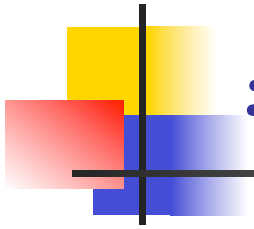


Digression: ? : operator

- C has one *ternary* operator (three arguments), the ? : ("question mark") operator
- Like an if statement that returns a value:

```
int i = 10;
int j;
j = (i == 10) ? 20 : 5; /* note 3 args */
/* "(i == 10) ? 20 : 5" means:
 * "If i equals 10 then 20 else 5." */
```

- Not used very often

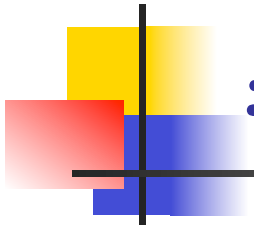


`#define` macros

- `#define` can also be used to define short function-like macros e.g.

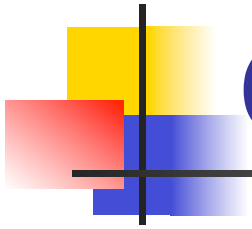
```
#define MAX(a, b) \
    (((a) > (b)) ? (a) : (b))
```

- Like a short function that gets expanded everywhere it's used (*a.k.a.* an *inline* function)
- But pitfalls exist (won't discuss further)



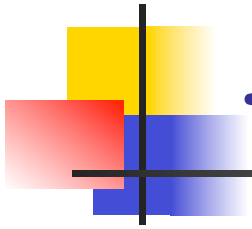
`#define` style

- `#define` defines new meaning for names
- Names that have been defined using `#define` are conventionally written with `ALL_CAPITAL_LETTERS`
- That way, they're easy to identify in code
- Conversely, don't use this style for regular variable names



Operators and precedence

- Low to high precedence:
  - `=` (assignment) `+=` `-=` `*=` `/=`
  - `==` `!=`
  - `<` `<=` `>` `>=`
  - `+` and `-`
  - `*` and `/`
  - `++` `--`
- 15 precedence levels in all!
- Use `( )` for all non-obvious cases

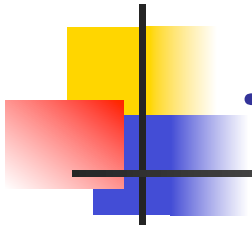


++ and -- (1)

- ++ and -- can be prefix or postfix

```
int a = 0;
a++;  /* OK */
++a;  /* OK */
```

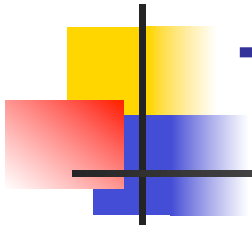
- Here they mean the same thing



++ and -- (2)

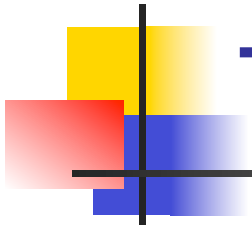
- Prefix is not the same as postfix!

```
int a, b, c;
a = 10;
b = ++a;  /* What is b? */
          /* 11 */
c = a++;  /* What is c? */
          /* 11 */
```



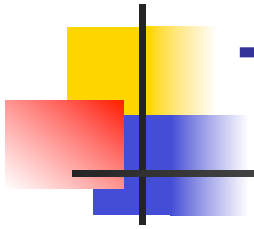
Types (1)

- `int`
  - usually 32 bits wide
  - could be 64 (depends on computer)
- `long`
  - "longer" integer
  - length >= length of `int`
  - usually same as `int`
- `short` (will see later in course)



Types (2)

- `float`
  - single-precision approximate real number
  - 32 bits wide
- `double`
  - double-precision
  - 64 bits wide



Type conversions (1)

- Converting numbers between types

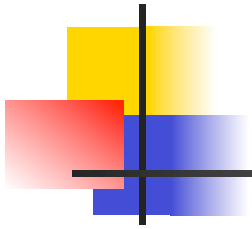
```
int i = 10;
float f = (float) i;
double d = (double) i;
```

- `(float)` etc. are type conversion operators
- Compiler will convert automatically
- But don't do it that way!

Type conversions (2)

- Dangers of implicit conversions:

```
int i, j;
double d;
i = 3;
j = 4;
d = i / j;                    /* d = ? */
                              /* 0.0 */
d = ((double) i) / ((double) j);
/* d = ? */
/* 0.75 */
```



Function prototypes (1)

- Normally, functions must be defined before use:

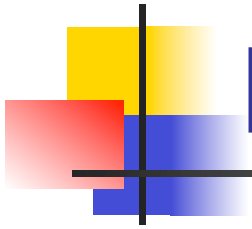
```
int foo(int x) { ... }
int bar(int y)
{
    return 2 * foo(y);
}
```

- Couldn't define **`bar`** before **`foo`**
- Compiler isn't that smart

Function prototypes (2)

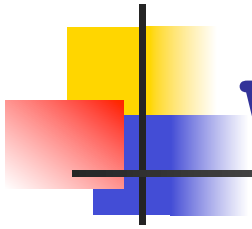
- Can get around this with function prototypes
- Consist of signature of function w/out body

```
int foo(int x);  /* no body yet. */
int bar(int y);  /* no body yet. */
int bar(int y)
{
    return 2 * foo(y);  /* OK */
}
 /* Define 'foo' later. */
```



Function prototypes (3)

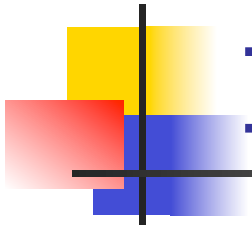
- Note that `foo` not defined when `bar` defined
- Rule of thumb: always write function prototypes at top of file
- That way, can use functions anywhere in file



while loops

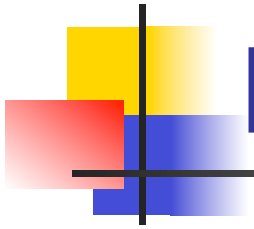
```
int a = 10;
while (a > 0)
{
    printf("a = %d\n", a);
    a--;
}
```

- Useful when # of iterations not known in advance



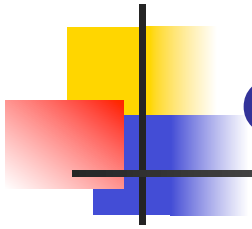
Infinite loops and **break**

```
int a;
while (1)  /*  or: for (;;)  */
{
     scanf("%d ", &a);
     printf("a = %d\n", a);
     if (a <= 0)
          break;  /* get out of loop */
}
```



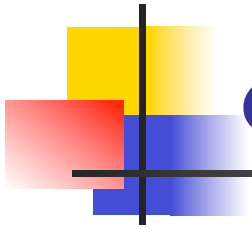
More on `break`

- `break` exits the nearest enclosing loop
- To exit more deeply-nested loops, need `goto`
- Avoid using `goto` in general



goto

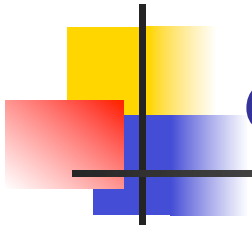
```
for (i = 0; i < m; i++) {
  for (j = 0; j < n; j++) {
      /* code ... */
      goto out; /* something went wrong */
  }
}
out:  /* a label */
/* continue here */
```



do/while

- Sometimes want to test at end of loop:

```
int i = 10;
do
{
        /* try something at least once */
        /* i gets changed */
}
while (i > 0);
```

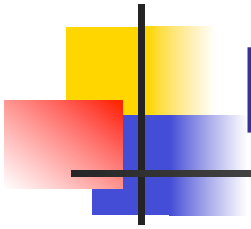


continue

- To exit a single iteration of a loop early, but keep on executing the loop itself, use a `continue` statement

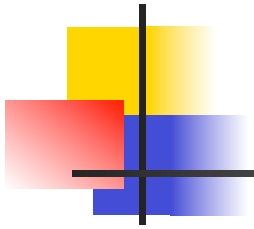
```
int i;
for (i = 0; i < 100; i++) {
    if (i % 2 == 0)
        continue;
    else
        printf("i = %d\n", i);
}
```

- Here, only prints out odd numbers



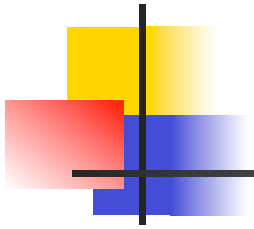
Note on syntax

- Body of `for`, `while`, `do`/`while`, `if`, `if`/`else` statements can be either
  - a block of code (surrounded by curly braces)
  - a single line of code
- Better to always use a block of code
  - expresses intent more clearly to reader
  - can add extra statements later more easily



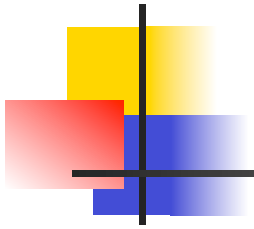
Input/output and `scanf()` (1)

- C provides three input/output "files" for you to use:
  - `stdin` for input from the terminal
  - `stdout` for output to the terminal
  - `stderr` for error output
    - normally also outputs to terminal
- All defined in `stdio.h` header file



Input/output and `scanf()` (2)

- `printf()` function outputs to `stdout`
- `scanf()` function reads from `stdin`
- More general versions to read from other files:
- `fprintf()` outputs to any file
- `fscanf()` reads from any file

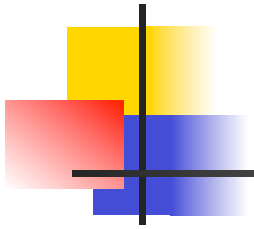


Input/output and `scanf()` (3)

- `fprintf()` and `stderr` used to print error messages:

```
fprintf(stderr,
        "something went wrong!\n");
```

- Still prints to terminal
- Always use this for printing error messages or program usage messages!

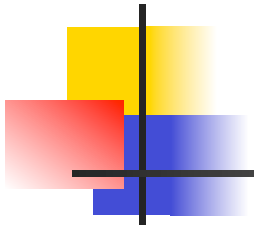


Input/output and `scanf()` (4)

- Recall `scanf()` function from lab 1
- Reads in from terminal input (known as `stdin`)
- Uses funny syntax *e.g.*

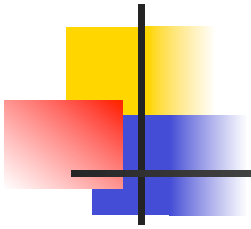
```
char s[100];

scanf("%99s", s);
```

- This says: "read in a string `s` that is no more than 99 characters long".



Input/output and `scanf()` (5)

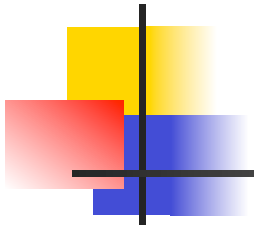
- `scanf()` changes the variable(s) in its argument list
- `scanf()` also returns an `int` value
  - if `scanf()` was successful, return the number of items read
  - if input unavailable, the special `EOF` ("end of file") value is returned
  - `EOF` is also defined in `stdio.h` header file



Input/output and `scanf()` (6)

- Testing `scanf()`'s return value:

```
int val;
int result;
result = scanf("%d", &val);
if (result == EOF)
{
    /* print an error message */
}
```

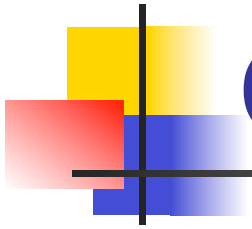


Input/output and `scanf()` (7)

- Notice the `&val` in the `scanf()` call:

```
int val, result;

result = scanf("%d", &val);
```

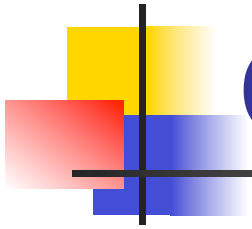
- What's that all about?
- Can't explain in detail now
- Will explain when we talk about pointers
- Rule: need `&` for reading `int` or `double`, but not strings



Commenting your code (1)

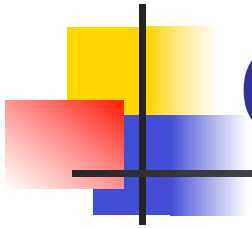
- The most important thing is to realize that

COMMENTS ARE VERY VERY IMPORTANT!



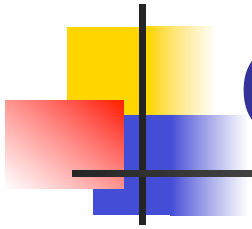
Commenting your code (2)

- Purposes of comments:
  - explain how to use your functions
  - explain how your functions work
  - explain anything that's tricky or non-obvious
- Who reads the comments?
  - anyone modifying your code
  - you, in a few weeks/months/years



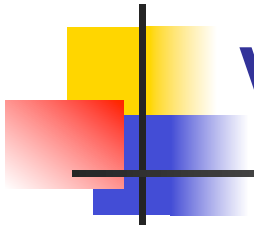
Commenting your code (3)

- Put comments right before functions
  - purpose of function
  - what arguments mean
  - what's returned
- Comment code that's not obvious
- Assume others will read your code
- Style (spelling, grammar) counts!
- Poor commenting ➔ marks off!



Good commenting

```
/*
 * area: finds area of circle
 * arguments: r: radius of circle
 * return value: the computed area
 */
double area(double r) {
    double pi = 3.1415926;
    return (pi * r * r);
}
```



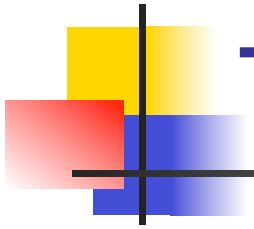
Variable names

- Usually use meaningful variable names

```
double x;  /* what does x mean? */
double distance;  /* better */
```

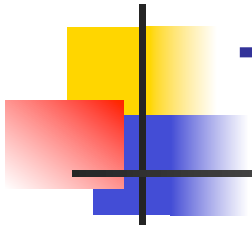
- Not always necessary

```
int loop_index;   /* bad */
int i;            /* good */
```



The `make` program (1)

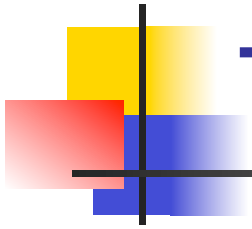
- `make` is a program which
  - automates compilation of programs
  - only recompiles files that
    - have changed
    - depend on files that have changed
- Only really useful for programs with multiple source code files



The `make` program (2)

- Write compilation info in a **`Makefile`**
- Usually compile by typing **`make`**
- Clean up by typing **`make clean`**
- We usually supply the **`Makefile`**
- Details:

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



The **make** program (3)

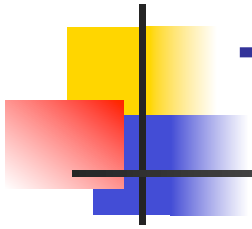
- Trivial **Makefile**:

```
program: program.o
	gcc program.o -o program
program.o: program.c program.h
	gcc -c program.c
clean:
	rm program.o program
```

The `make` program (4)

- Targets in red

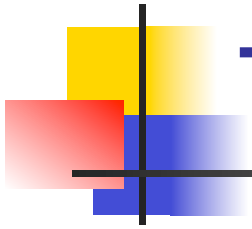
```
program: program.o
	gcc program.o -o program
program.o: program.c program.h
	gcc -c program.c
clean:
	rm program.o program
```



The `make` program (5)

- Dependencies in green

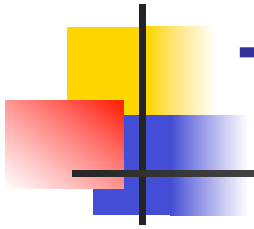
```
program: program.o
	gcc program.o -o program
program.o: program.c program.h
	gcc -c program.c
clean:
	rm program.o program
```



The `make` program (6)

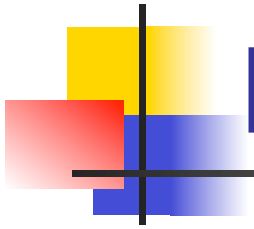
- Commands in blue

```
program: program.o
	gcc program.o -o program

program.o: program.c program.h
	gcc -c program.c

clean:
	rm program.o program
```



The `make` program (7)

- If `program.c` or `program.h` changes
  - `program.o` is now out-of-date
  - `program.o` gets recompiled (changes)
  - `program` is now out-of-date
  - `program` gets recompiled
- If multiple `.c` files exist and only one changes, only necessary files recompiled



Next week

- Arrays
- Strings
- Command-line arguments
- `assert`