C Tutorial
Session #2

- Type conversions
- More on looping
- Common errors
- Control statements
- Pointers and Arrays
- C Pre-processor
- Makefile
- Debugging

Type Conversions

```
main()
{
int i;
unsigned int stop_val;

stop_val = 0;
i = -10;

while (i <= stop_val)
   {
   printf ("%d\n", i);
   i = i + 1;
   }
}
```

Common mistake

- Example:

```
int n = 0;
while (n < 3)
    printf(" n is %d\n", n);
    n++;
```

Troubles with Truth

```
status = scanf("%ld", &num);
while (status = 1) {
  printf("please enter a number\n");
  status = scanf("%ld", &num);
}
```

The *for* loop

- Syntax:

  for (exp1; exp2; exp3)

- Example:

```
for (count = 1; count < 10; count++)
    {
    printf("count is %d\n", count);
    }
```

Count by charaters

```
char count;
for (count = 'a'; count <= 'z'; count++) {
  printf("count is %c\n", count);
}
```

Nested loop

Example:

```
for (row = 0; row < 10; row++) {
    for (ch = 'a'; ch  < 'f'; ch++) {
          printf("%c ", ch);
    }
}
```

C Control Statements

- *if* statement
- Syntax:

```
if (expression) {
  statements
}
else {
  statements
}
```

Sample code

```
if (x > 0) {
   printf(" x is greater than 0.\n");
} else {
  printf(" x is less than or equals to
  0.\n");
}
```

Another example

```
int main() {
   char ch;
   ch = getchar();
   while (ch != '\n') {
       putchar(ch);
       ch = getchar();
   }
   return 0;
}
```

Using continue and break

```
ch = getchar();

while (ch != '\n')
    {
     if (ch == 'q')
        break;
     if (ch == '$')
        {
        printf("dollar!\n");
        continue;
        }
    putchar(ch);
    ch = getchar();
    }
printf("finished.\n");
```

Continue and break

- *continue* skips the rest of current iteration and starts the next iteration.
- *break* causes the program to break free of the loop that encloses it and to proceed to the next stage of the program.

Multiple choice: switch and break

```
switch (ch)
{
   case '\n':
      printf("newline\n");
      break;
   case 'q':
      exit(0);
   case '$':
      printf("dollor!.\n");
      break;
   default:
      putchar(ch);
      break;
}
```

Arrays

- An array is composed of a series of elements of one data type.
- An array declaration tells the compiler how many elements the array contains and what the type is for these elements.
- Example:

```
float candy[365]; char code[12];
```

Assigning Array Values

- Initialization.

```
int arr[6] = {0, 1, 2, 3, 4, 5};
```

- Assigning values.

```
for (counter = 0; counter < 6; counter++) {
  arr[counter] = counter;
}
```

Multidimensional Arrays

- An two dimension array is an array of one dimension arrays.
- Example: `float rain[5][12];`

  rain is an array of five elements. Each element is an array of twelve float point numbers.

Pointers and Arrays

- Pointers provide a symbolic way to use address.
- Array notation is simply a disguised use of pointers.

```
rain = &rain[0] // true
```

Functions, Arrays and Pointers

- Suppose you want to write a function that operates on an array. What would the function call look like?

```
total = sum(marble);
```

- What would the function declaration be?

```
int sum(int *ar);
```

- NOTE: array name is the address of the first elements.

Pointer Operations

- Example:

```
int urn[5] = {1, 2, 3, 4, 5};
int *ptr1, *ptr2, *ptr3;
ptr1 = urn;
ptr2 = &urn[2];
```

- Question:
  if ptr1[x] == ptr2[1] is true, what is x?

C Preprocessor

- #include <string.h> (or “string.h”)
  There is a difference between the < > -- look for the file in the include path and “ “ look first in local directory, then in the include path
- #define [name]
  #define [name] value
  When [name] is used it is replaced with its value
  Eg. #define DEBUG
  #define DEBUG 1
- #ifdef [name]
  #ifndef [name]
- #endif

Examples

```
#ifdef DEBUG
printf ("entering main
()\n");
#endif
…
#define TRUE 1
#define FALSE 0
#define BYTE unsigned char
BYTE success;

success = function ();
```

```
if (success == FALSE);
   {
#ifdef DEBUG
   printf ("error\n”);
#endif

   exit (0);
   }
```

Makefile

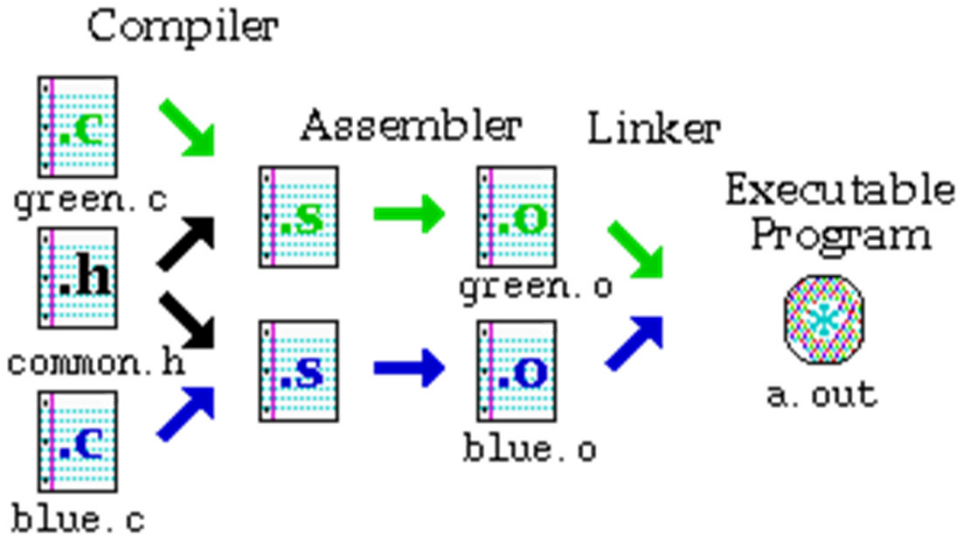
- The **make** command allows you to manage large programs or groups of programs.
- The **make** program keeps track of which portions of the entire program have been changed, compiles only those parts of the program which have changed since the last compile.

Compilation Steps

1. **Compiler** stage: All C language code in the *.c* file is converted into a lower-level language called Assembly language.
2. **Assembler** stage: The assembly language code is converted into *object code*.
3. **Linker** stage: The final stage in compiling a program involves linking the object code to code libraries and produces an executable file.

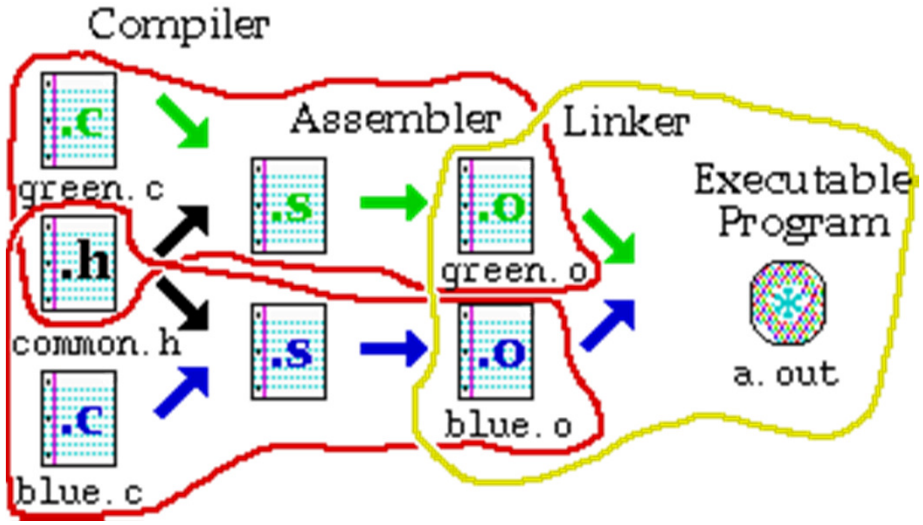
Compile Multiple files

- Programmers usually divide source code into separate easily-manageable *.c* files when programs become large.



Separate Compilation Steps

- Compile *green.o*: cc -c green.c
- Compile *blue.o*: cc -c blue.c
- Link the parts together: cc green.o blue.o



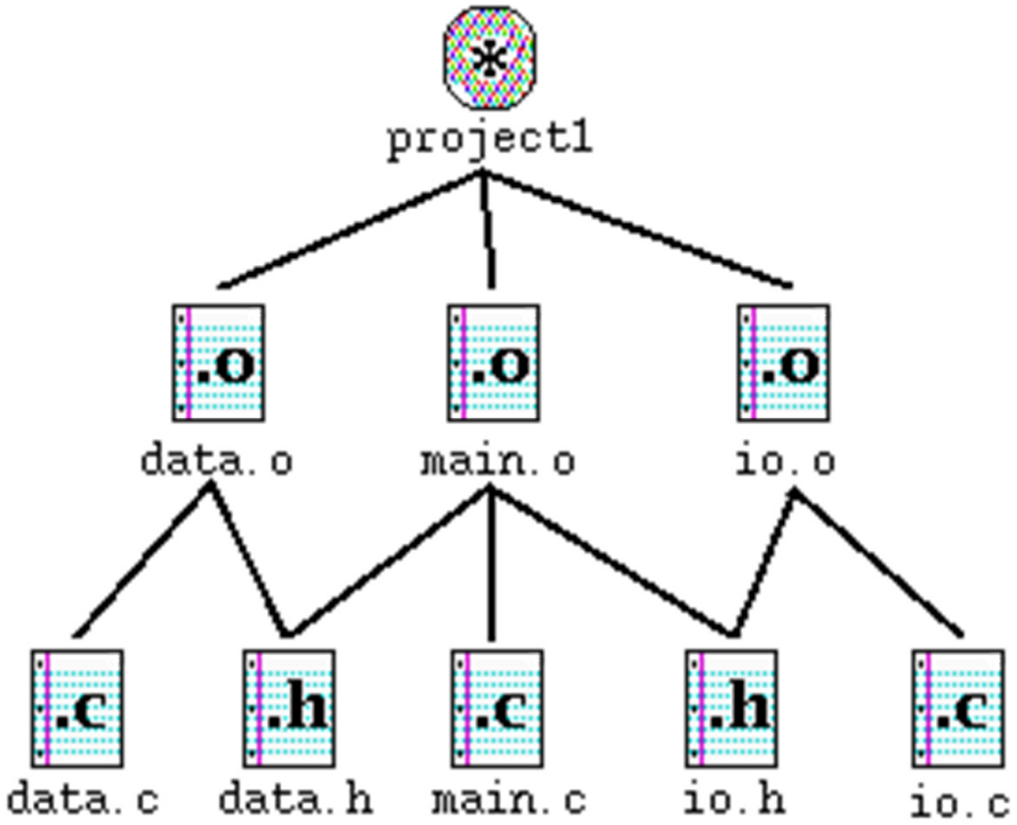
Using multiple source files

- no two files have functions with the same name in it.
- no two files define the same global variables.
- To use functions from another file, make a *.h* file with the function prototypes, and use #include to include those *.h* files within your *.c* files.
- At least one of the files **must** have a main() function.

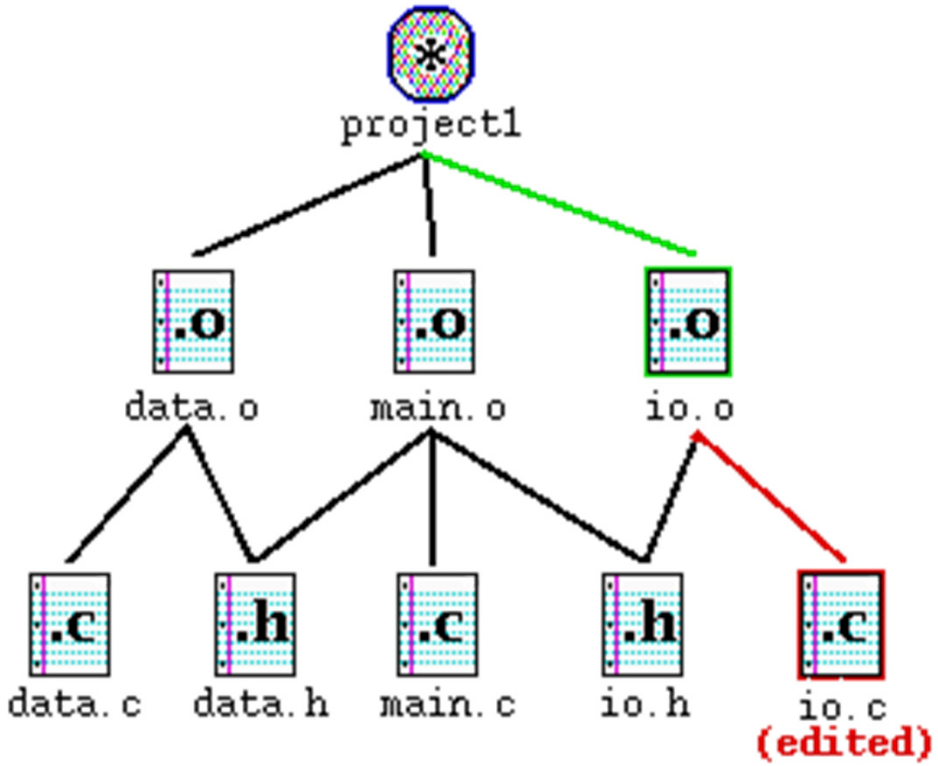
Dependencies

- Make creates programs according to the file dependencies.

Dependency graphs



How dependency works

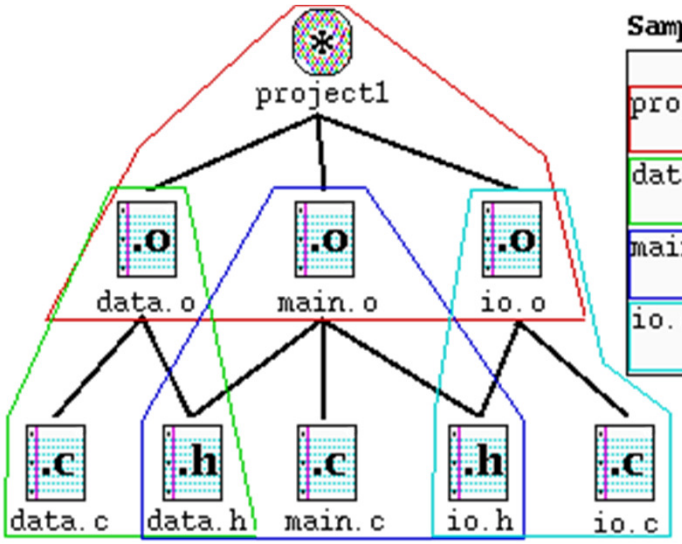


How does make do it?

Sample Makefile

```
project1: data.o main.o io.o
    cc data.o main.o io.o -o project1
data.o: data.c data.h
    cc -c data.c
main.o: data.h io.h main.c
    cc -c main.c
io.o: io.h io.c
    cc -c io.c
```

Translating the dependency graph



Sample Makefile

```
project1: data.o main.o io.o
    cc data.o main.o io.o -o project1
data.o: data.c data.h
    cc -c data.c
main.o: data.h io.h main.c
    cc -c main.c
io.o: io.h io.c
    cc -c io.c
```

target : source file(s)

command *(must be preceded by a tab)*

Using the Makefile with make

- Once you have created your *Makefile* and your corresponding source files, you are ready to use it by typing **make**.

Basic Debugging

- Prepare code for debugger
  add –g option to compiler command
  gcc –g –o <file> <file.c>
- Debug the program
  gdb <file>
- Breakpoints and execution
  b <line#> OR <function>
  s – step
  c – continue
  n – next
- Look at variables
  print <expr>