Lecture 2:
More C Programming

CS211: Computer Architecture
Summer 2020

Logistics

- Sakai Meetings (aka BigBlueButton)
  - If you had any issues for this lecture, email me
- iLab Machine and C Programs
  - Make sure by now you start on figuring out iLab machines
  - C programs have been provided in resources to help you get going
- Piazza
  - Make sure you have signed up if you haven't already at on Sakai -> "Piazza"
  - You can set it up so you can get email digests of activity on piazza
- P/NC policy extended to Summer Session
  - https://nbprovost.rutgers.edu/grade-change-academic-deadlines
  - https://nbprovost.rutgers.edu/guidance-faq
- Project 1 will be released by next lecture
- Mistake in previous lecture:
  - %lf is the formate specifier for double (essentially long float)

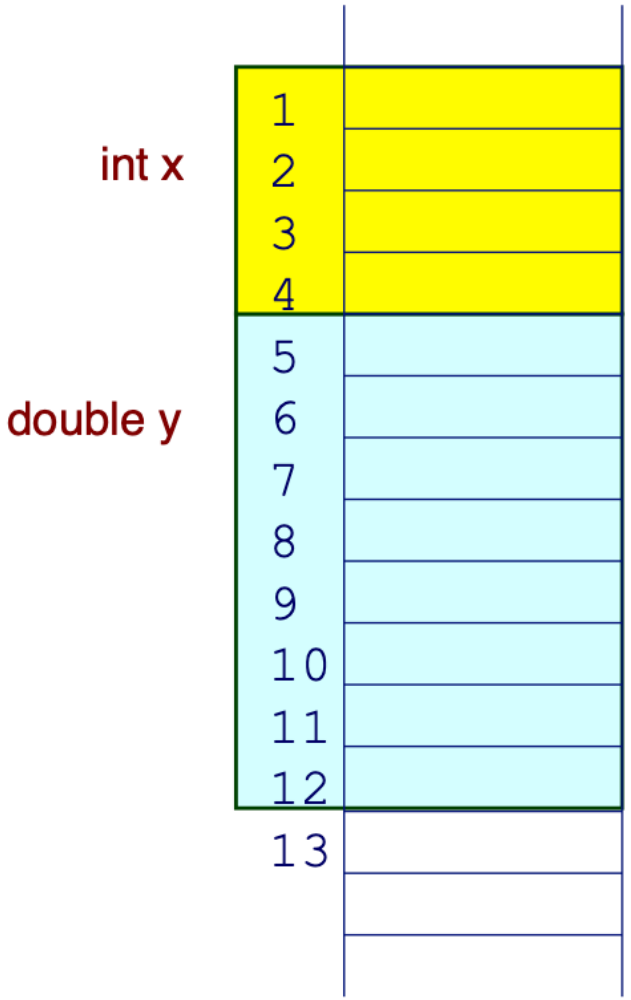
C Programming (part 2)

Topics

- Intro to memory (Addressing)
- Pointers
  - Pass by value vs pass by reference
  - Pointer Arithmetic
- Arrays
- Strings
- Structs

Intro to Memory

- C's memory model matches the underlying (virtual) memory system
  - Array of addressable bytes
- Variables are simply names for contiguous sequences of bytes
  - Number of bytes given by type of variable
- Compiler translates names to addresses
  - handles memory addressing for us



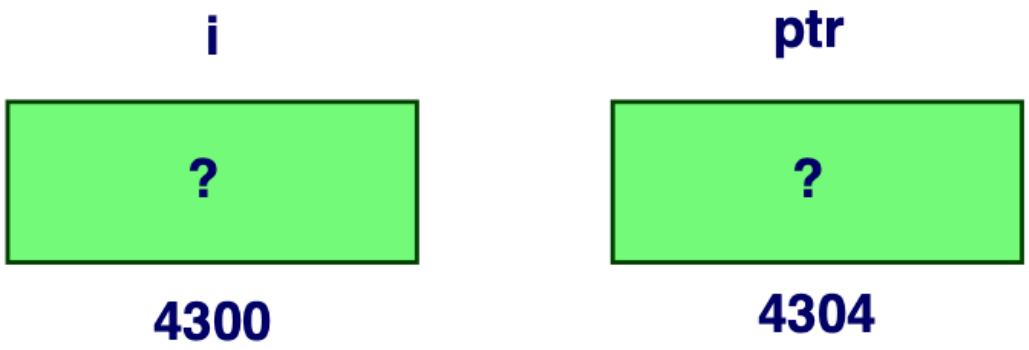
Pointers

- A pointer simply an address
- Pointer Declaration
  - Data type followed by *
  - Example:
    - int *p; // p will point to an int
    - You can think of it as ”variable p will hold the address of a int”
- Pointer Operators
  - & (Address)
    - &X gives us the address of variable X
  - * (De-referencing)
    - *X gives value stored at the address X

Pointer (example)

```
int i;
int *ptr;

i = 4;
ptr = &i;
*ptr = *ptr + 1;
```

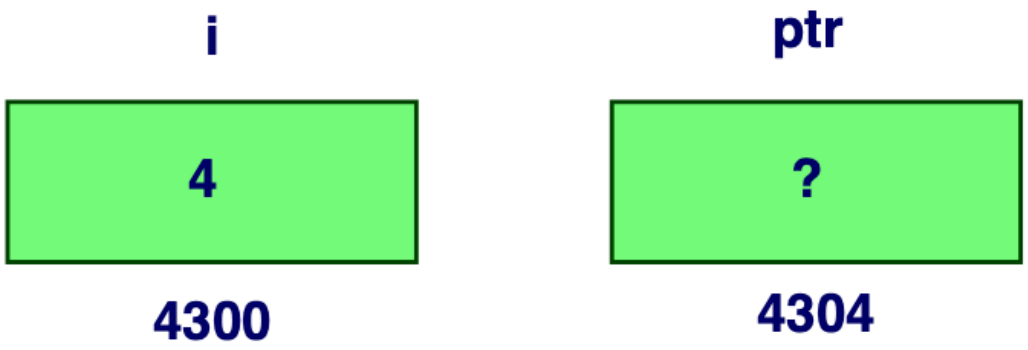


Pointer (example)

```
int i;
int *ptr;

i = 4;
ptr = &i;
*ptr = *ptr + 1;
```

store the value 4 into the memory location associated with i

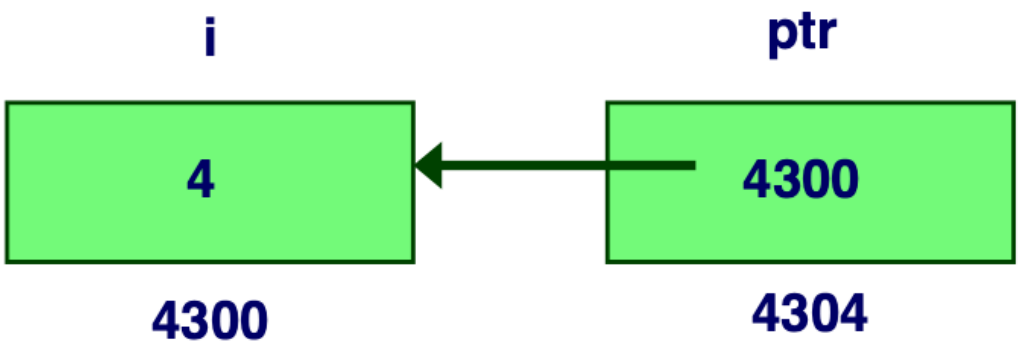


Pointer (example)

```
int i;
int *ptr;

i = 4;
ptr = &i;
*ptr = *ptr + 1;
```

store the address of i into the memory location associated with ptr



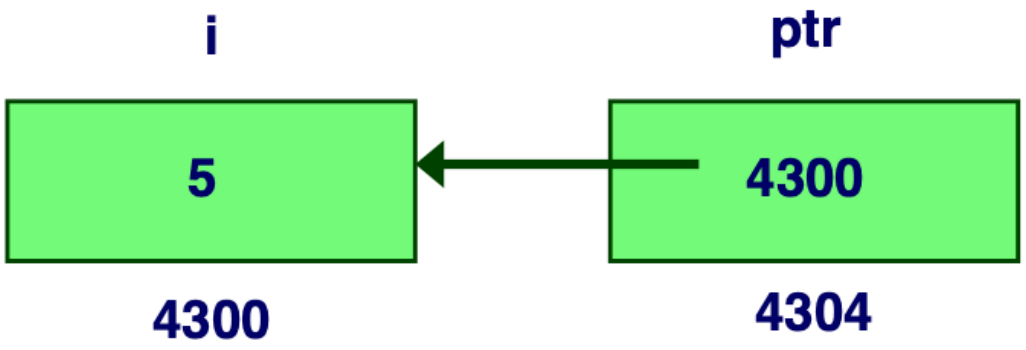
Pointer (example)

```
int i;
int *ptr;

i = 4;
ptr = &i;
*ptr = *ptr + 1;
```

store the result into memory at the address stored in ptr

read the contents of memory at the address stored in ptr



Parameter Passing

- Pointers give us an additional way of passing parameters
- Two ways of passing parameters to functions
- Pass by Value
  - Simply give the pass of the value of a variable
  - Example:
    - int x = 5;
SomeFunction(x);
    - The function has a copy of x's value, 5
- Pass by Reference
  - Pass a pointer to a location
  - Example:
    - int x = 5;
int *y = &x;
function(y);
    - The function has the pointer to where x is stored
    - function manipulate what value is stored where x is

Pass by Value (example)

- What does the following code produce? Why?

```
void Swap(int firstVal, int secondVal)
{
    int tempVal = firstVal;
    firstVal = secondVal;
    secondVal = tempVal;
}

…
int fv = 6, sv = 10;
Swap(fv, sv);
printf("Values: (%d, %d)\n", fv, sv);
```

Pass by Value (example)

- What does the following code produce? Why?

```
void Swap(int firstVal, int secondVal)
{
    int tempVal = firstVal;
    firstVal = secondVal;
    secondVal = tempVal;
}

…
int fv = 6, sv = 10;
Swap(fv, sv);
printf("Values: (%d, %d)\n", fv, sv);
```

Answer: 6, 10

Pass by Reference

- What does the following code produce? Why?

```
void Swap(int *firstVal, int *secondVal)
{
    int tempVal = *firstVal;
    *firstVal = *secondVal;
    *secondVal = tempVal;
}

…
int fv = 6, sv = 10;
Swap(&fv, &sv);
printf("Values: (%d, %d)\n", fv, sv);
```

Pass by Reference

- What does the following code produce? Why?

```
void Swap(int *firstVal, int *secondVal)
{
    int tempVal = *firstVal;
    *firstVal = *secondVal;
    *secondVal = tempVal;
}

…
int fv = 6, sv = 10;
Swap(&fv, &sv);
printf("Values: (%d, %d)\n", fv, sv);
```

Answer: 10, 6

Null Pointer

- Sometimes we want to know a pointer points to nothing
  - Example: So we don't try and access an invalid location or a pointer we haven't set yet
- Setting a pointer to Null
  - Use the NULL constant
  - Example:
    - int *p;
      p = NULL;
- NULL is a predefined constant that contains a value that a non-null pointer should never hold
  - Often, NULL = 0, because address 0 is not legal for most platforms

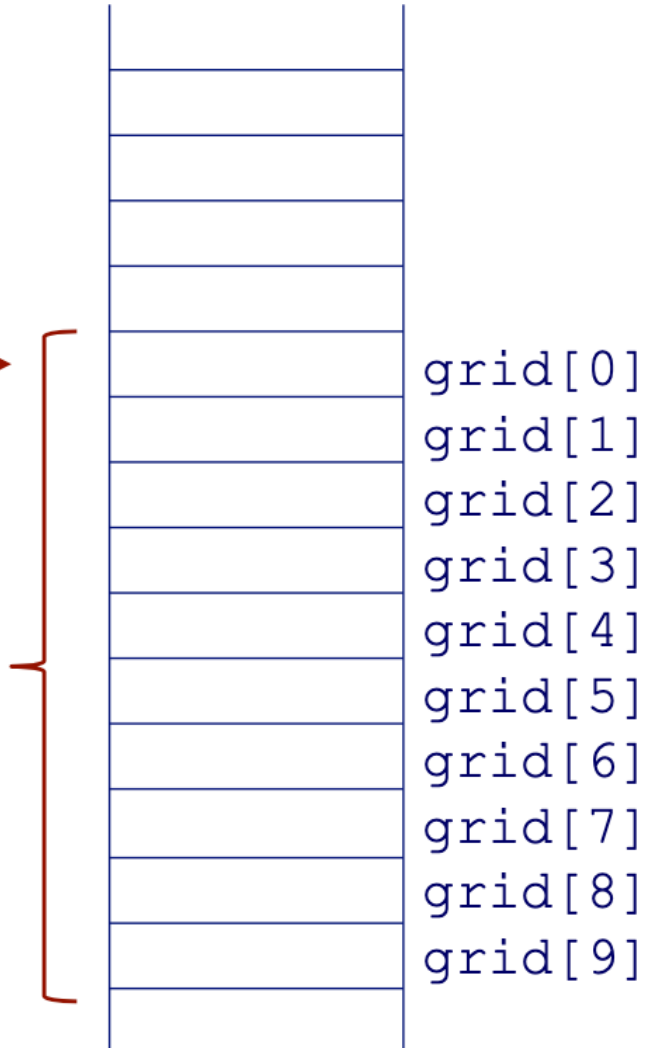
Arrays

- Arrays are contiguous sequences of data items
- All data items are of the same type
- Declaration of static array
    - Example:
        - int x[10];
        - int y[10] = {1,2,3,4,5,6,7,8,9,0};
- Array index always starts at 0
- The C compiler and runtime system do not check boundaries
    - The compiler will happily let you do the following:
        - int a[10];
          a[12] = 5;

Array Storage

- Elements of an array are stored sequentially in memory
- First element (grid[0]) is at the lowest address of sequence

char grid[10];



- Variable grid is simply address of the beginning of sequence
- Knowing the location of the first element is good enough to access any element
  - Can access any element using starting address, index, and size of each element
  - Address of array element would simply be: starting address + (element size * index)

Arrays and Pointers

- An array name is essentially a pointer to the first element in array
- We can do the following:

```
char word[10];
char *cptr = word;
```

- Getting first element would be
  - char x = word[0];

    or

    char x = *cptr;
- Getting the 5th element would be
  - char x = word[4];

    or

    char x = *(cptr + 4) (we'll go over why this is in the next slide)

Pointer Arithmetic

- We can manipulate pointers and calculate addresses by using pointer arithmetic
- Address calculations with pointers are dependent on the size of the data the pointers are pointing to
- Example:
  - size of type int is 4 bytes
  - ```
    int *i;
    i++;        // equivalent to i = i + 4
    i--;        // i = i - 4
    i += 2;     // i = i + (2 * 4)
    ```
- Another example:
  - ```
    int x[10];
    int *y = x;
    *(y + 3) = 13       /* equal to x[3] = 13 */
    ```

Passing Arrays as Arguments

- Arrays are passed by reference
  - Ex. function(array);
- Array items are passed by value
  - Ex. function(array[10]);
- What will be the result?

```
void foo(int nums[], int x){
        nums[0] = 5;
        x = 5;
}

int main(int argc, char **argv){
        int z[2];
        z[0] = 0;
        z[1] = 0;
        foo(z, z[1]);
        printf("First number is %d, Second number is %d\n",  z[0],  z[1]);
}
```

Passing Arrays as Arguments

- Arrays are passed by reference
  - Ex. function(array);
- Array items are passed by value
  - Ex. function(array[10]);
- What will be the result?

```
void foo(int nums[], int x){
        nums[0] = 5;
        x = 5;
}

int main(int argc, char **argv){
        int z[2];
        z[0] = 0;
        z[1] = 0;
        foo(z, z[1]);
        printf("First number is %d, Second number is %d\n",  z[0],  z[1]);
}
```

Answer: 5, 0

Common Pitfalls with Arrays in C

- Overrun array limits
  - There is no checking at compile or run time to see whether you are within array bounds
    Ex.

```
int array[10];
for (int i = 0; i <= 10; i++){
        array[i] = 0;
}
```

- No such thing as declaration with variable size
  - Size of array must be known at compile time
  - The following will not work:
    - int array[x];

Strings

- String are simply an array of characters
  - Can allocate like any other array
    char str[10]:
- Each string should end with a '\0' characters
  - '\0' (aka null terminator) lets the computer know that the string has ended
  - Functions like strlen() need this to work on arbitrary strings
  - Make sure there is enough space for the null terminator
- Special syntax for initializing string:
  - char str[16] = "Result";
  - equivalent to:

```
str[0] = 'R';
str[1] = 'e';
str[2] = 's';
….
str[6] = '\0';
```

Useful String Functions

- String library part of the standard C libraries
  - #include <string.h>
- size_t strlen(char *s)
  - computes the length of string s
- char *strcpy(char *dest, char *src)
  - copies string from src to dest
- int strcmp(char *str1, char * str2)
  - Compares str1 to str2

Structs

- A struct is a mechanism for grouped together related data items
- Example:
  - We might want to represent an airborne aircraft:

```
char flightNum[7];
int altitude;
int longitude;
int latitude;
int heading;
double airSpeed;
```

Declaring a Struct Type

- Define a new struct type to tell the compiler what our struct will look like:

```
struct flightType{
        char flightNum[7];
        int altitude;
        int longitude;
        int latitude;
        int heading;
        double airSpeed;
};
```

- This tells the compiler what how big our struct is how the different data items are laid out in memory
- Declaration doesn't allocate any memory yet, memory will allocated when a variable of struct flight is declared

Using a Struct

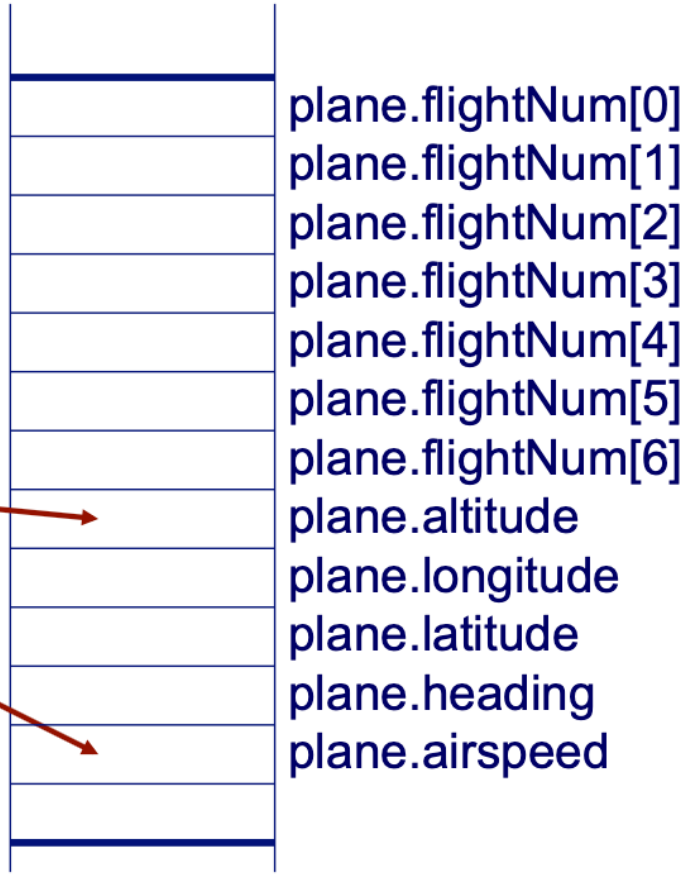
- To use a struct, declare a variable using the new struct type

  Ex. struct flightType plane;

- We can access the different members like the following

```
plane.altitude = 10000;
plane.airSpeed = 800.0;

foo(&(plane.airSpeed);
/* pass the address of
   plane.airSpeed */
```



Array of Structs

- We can also declare arrays of struct items
  - struct flightType planes[100];
- Each array element is an item of type ”struct flightType”
- To access members of a particular element:
  - planes[34].altitude = 10000;
- Because the [] and . Operators have the same precendence, this is the same as:
  - (planes[34]).altitude = 10000;

Pointers to Structs

- We can declare and create a pointer to a struct:
  - struct flightType *planePtr;
    planePtr = &planes[34];
- To access a member of a struct addressed by a pointer:
  - (*planePtr).altitude = 10000;
- Because the operator has a higher precendence than *, this is NOT the same as:
  - *planePtr.altitude = 10000;
- Luckily C provides special syntax for accessing a struct member through a pointer:
  - planePtr->altitude = 10000;

Passing Struct as Arguments

- It is possible to pass a struct by value however is not recommended
- Most of the time, you'll want to pass a pointer to a struct

```
int willcollide(struct flightType * planeA, struct flightType *planeB){

        if(planeA->altitude == planeB->altitutde){
                return 1;
        }else{
                return 0;
    }

}
```

Compilation, Make, and Makefiles

C Program Compilation

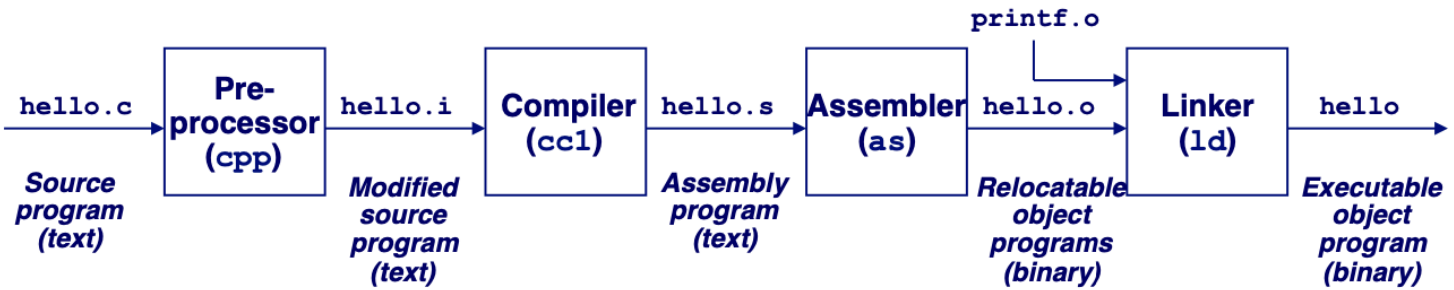
- Compile
  - gcc HelloWorld.c -o HelloWorld
  - Results in binary executable named HelloWorld
- Running it:
  - ./hello
- What if our programs become complex and there are multiple .c files for our program?

C Program Compilation

- To compile a program divded across multiple source files we can simply
  - gcc file1.c file2.c file3.c ….. fileN.c -o program
- This would work, but it may take a while
- If there is change in one of the files, all files need to be recompiled
- When dealing with large code bases, this won't be great

More Efficient Compiling

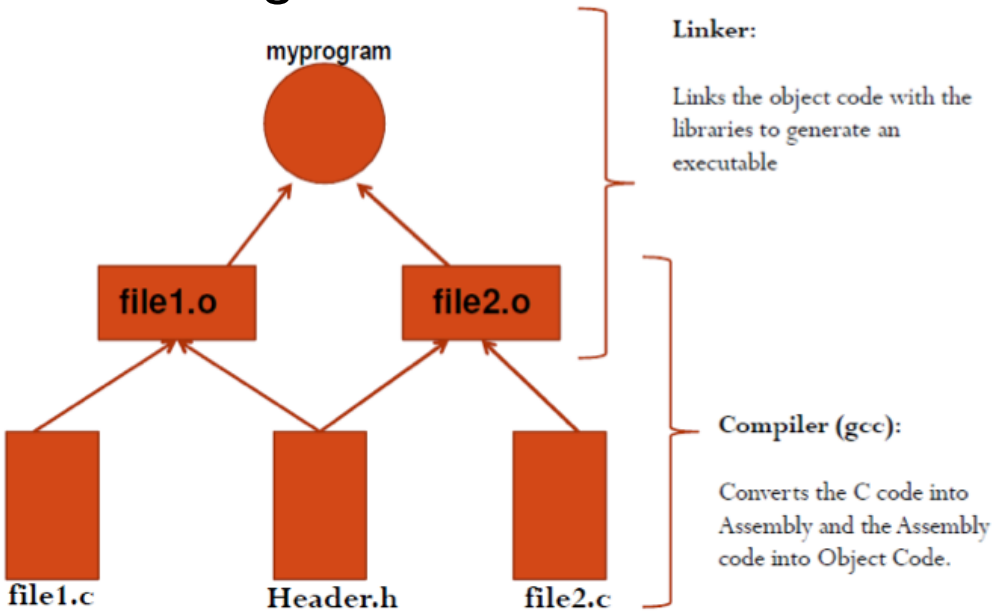
- Recall the compilation progress



- .c files are compiled and assembled into objects then linked to created the final executable object
- We can generate object files in isolation and link them together

More Efficient Compiling

- Consider the following:



- Generate file1.o and file2.o then link them
  - gcc -c file1.c (This will generate file1.o)
  - gcc -c file2 .c (This will generate file2.o)
  - gcc -o program file1.o file2.o (Link and generate executable named "program")
- If we change one file, we only need to compile that one file's object file (.o) and link it with the existing object files
- This would save processing time every time we want to compile

Make

- Make is a utility and is used to help compile programs
- Helps automates compilation process for large c projects
- It is good practice to use as you develop bigger and more complex code

Makefile

- In order to use Make you need to have a makefile
- A makefile is file that consists of various "rules" that consists of commands to run
- Format for rules:

```
<rulename>: <dependencies>
        <commands to run>
```

Note: command lines after the rule must be tabbed, no spaces or else make will complain

- Example:

```
all: program.c
        gcc -o program program.c
```

- If we run *make <rulename>*, make
  - makes sure the dependencies are met
  - if some dependencies are not met, then run rules to generate those dependencies if possible
  - Once dependencies are met then run the commands

Simple Makefile Example

- Example makefile

```
all: program.c
	gcc -o program program.c

clean:
	rm -rf program
```

- Typically there are two common rules that you should define
  1. all rule - main rule that's ran when running just "make"
  2. clean rule - rule to clean up all generated files
- For the example when we run "make", Make
  1. Looks if program.c exists
  2. If it does, run command "gcc -o program program.c"
- For the example when we run "make clean"
  1. Runs "rm -rf program" which deletes the program executable

More Typical Makefile Example

- Example makefile for a program with multiple files:

```
all: program

program: file1.o file2.o main.o
	gcc -o program file1.o file2.o main.o

clean:
	rm -rf file1.o file2.o program
```

- Where are the rules to generate the object files (.o)?
- Luckily Make automatically finds the corresponding .c files for a .o file and generates the .o file to resolve object file dependencies
- Make will also NOT recompile an object file if it has been previously compiled and the corresponding source code (.c file) has not changed
  - Saves us time by compiling only the code that changed

- Learn More:
  - https://www.gnu.org/software/make/manual/make.html
- Try and make your own makefiles for your own programs