CSE 1320

Week of 02/27/2023

Instructor : Donna French

Size of Pointers

```
char *cPtr = NULL;
int *iPtr = NULL;
long long *llPtr = NULL;
float *fPtr = NULL;

printf("%ld\n", sizeof(cPtr));
printf("%ld\n", sizeof(iPtr));
printf("%ld\n", sizeof(llPtr));
printf("%ld\n", sizeof(fPtr));
```

```
8
8
8
8
```

Size of Pointers

```
char cVar = 'A';
cPtr = &cVar;
int iVar = INT_MAX;
iPtr = &iVar;
long long llVar = LLONG_MAX;
llPtr = &llVar;
float fVar = FLT_MAX;
fPtr = &fVar;

printf("%ld\n", sizeof(cPtr));
printf("%ld\n", sizeof(iPtr));
printf("%ld\n", sizeof(llPtr));
printf("%ld\n", sizeof(fPtr));
```

```
8
8
8
8
```

Why do pointers need to have a type?

For the same reason that languages have a type system in the first place - it helps to detect invalid usage of pointers at compile time rather than at runtime.

If you pass a pointer to a char to a function that expects a pointer to an int, not only is the result of that function not going to make any sense, but it will be invoking undefined behavior because the function will try to read sizeof(int) bytes, and you've only supplied 1.

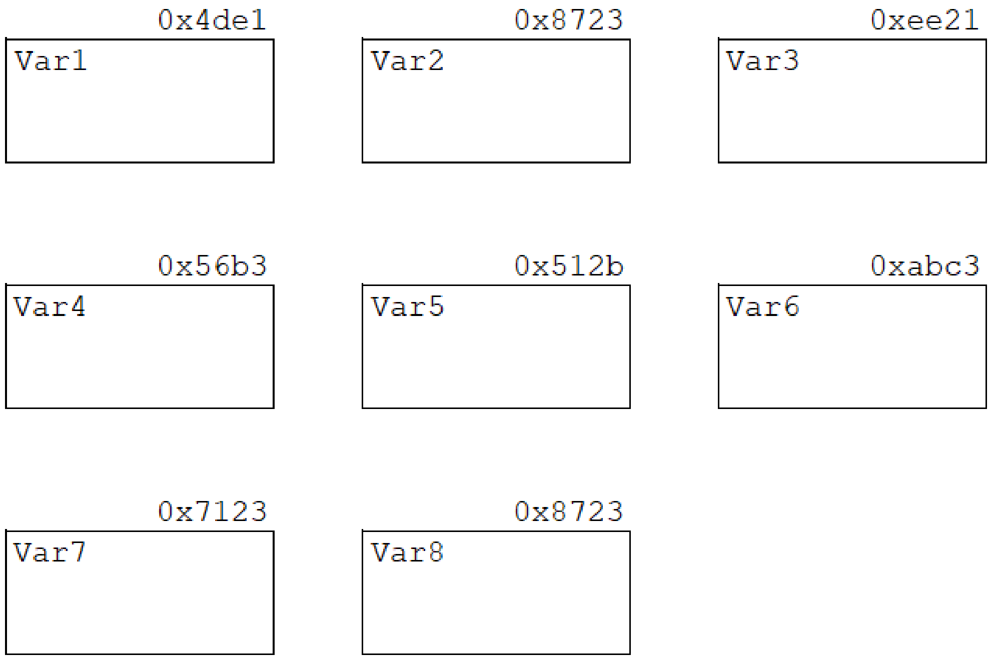
If your pointers have a type, however, the compiler will generate an "passing argument from incompatible pointer type" error (this is sometimes a warning rather than an error, depending on your compiler).

Why do pointers need to have a type?

The other major difference is that when you do pointer arithmetic, the units you're working in are equal to the size of the element that that pointer points to.

So if you allocate a pointer to an array, and then add 3 to it, the result will be a pointer to the 4th element in the array, regardless of the type of the elements.

```
int Var1 = 50;
int Var2 = 100;
int Var3 = 200;

int *Var4 = &Var1;
int *Var5 = &Var2;
int *Var6 = &Var3;

int **Var7 = &Var5;
int ***Var8 = &Var7;
```



0x4de1 Var1

0x8723 Var2

0xee21 Var3

0x56b3 Var4

0x512b Var5

0xabc3 Var6

0x7123 Var7

0x8723 Var8

```
printf("1. %d\n", Var1);                 1. 50

printf("2. %d\n", Var2+*Var5);           2. 200

printf("3. %d\n", Var3-**Var7);          3. 100

printf("4. %d\n", *&Var3);               4. 200

printf("5. %d\n", *Var4+Var1);           5. 100

printf("6. %d\n", *Var5);                6. 100

printf("7. %d\n", *Var6-*&*&Var1);       7. 150

printf("8. %d\n", **Var7);               8. 100

printf("9. %ld\n", *Var7 - Var5);        9. 0

printf("10. %ld\n", Var7 - *Var8);       10.0
```

The `do-while` Statement

```
do
{
    statement
}
while (expression);
```

Looping structure

- `statement` will always execute at least once
- `expression` will be evaluated after `statement` executes
- loop repeated if `expression` is nonzero
- a semicolon is required after `expression`

The `do-while` Statement

```c
int AskAgain = 1;

while (AskAgain)
{
   printf("Please enter a decimal "
          "number between 0 and 255 ");
   scanf("%d", &DecNum);

   if (DecNum >= 0 && DecNum <= 255)
      AskAgain = 0;
   else
   {
      AskAgain = 1;
      printf("\nYou entered a number “
             "not between 0 and 255\n\n");
   }
}
```

```c
int AskAgain;

do
{
   printf("Please enter a decimal "
          "number between 0 and 255 ");
   scanf("%d", &DecNum);

   if (DecNum >= 0 && DecNum <= 255)
      AskAgain = 0;
   else
   {
      AskAgain = 1;
      printf("\nYou entered a number “
             "not between 0 and 255\n\n");
   }
}
while (AskAgain);
```

Can we create a better while loop?

```
int AskAgain = 1;

while (AskAgain)
{
   printf("Please enter a decimal "
          "number between 0 and 255 ");
   scanf("%d", &DecNum);

   if (DecNum >= 0 && DecNum <= 255)
      AskAgain = 0;
   else
   {
      AskAgain = 1;
      printf("\nYou entered a number “
            "not between 0 and 255\n\n");
   }
}
```

```
printf("Please enter a decimal "
       "number between 0 and 255 ");
scanf("%d", &DecNum);

while (DecNum < 0 || DecNum > 255)
{
   printf("\nYou entered a number "
          "not between 0 and 255\n\n");
   scanf("%d", &DecNum);
}
```

Can we create a better do-while loop?

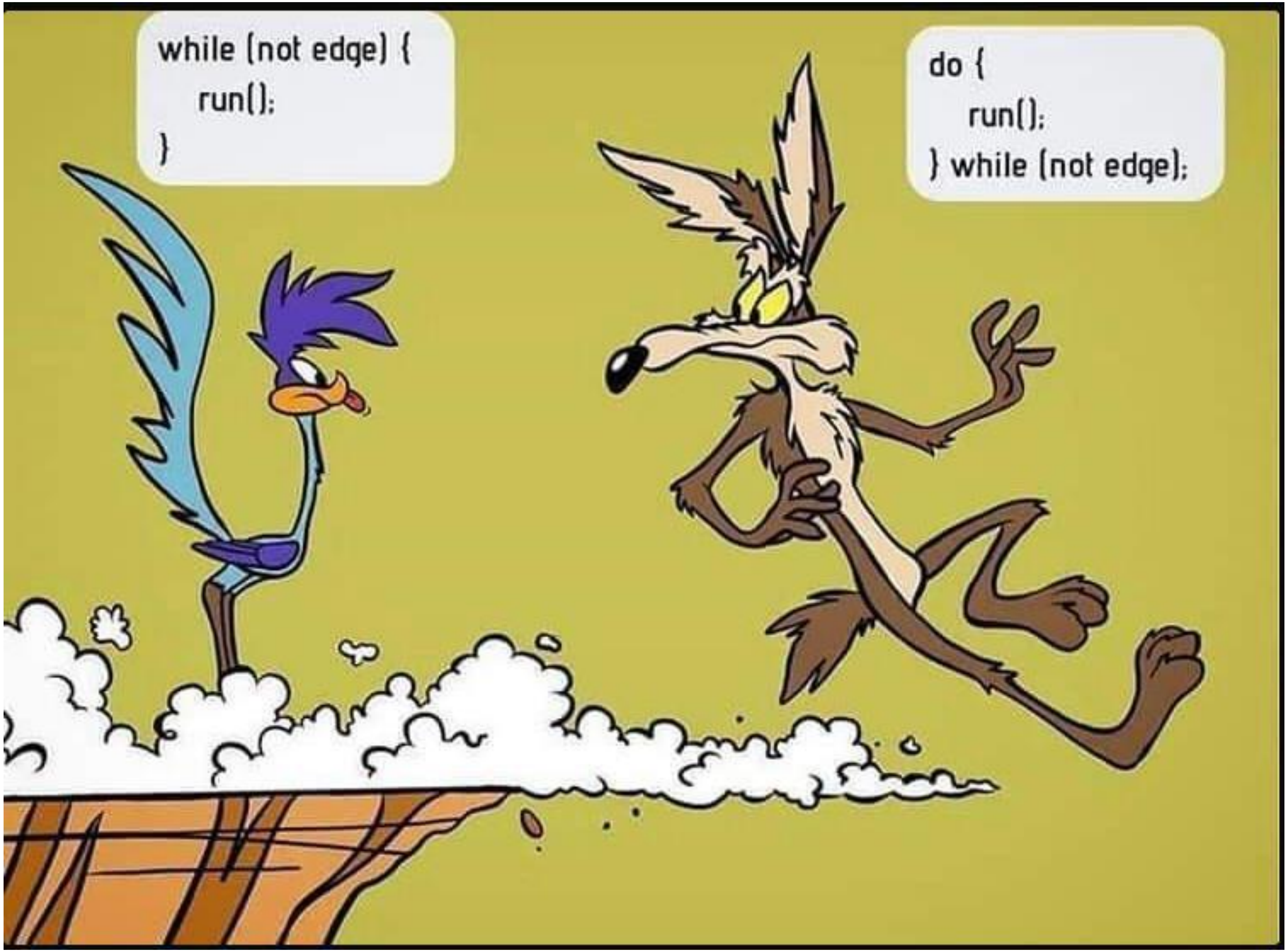
```
int AskAgain;

do
{
   printf("Please enter a decimal "
          "number between 0 and 255 ");
   scanf("%d", &DecNum);

   if (DecNum >= 0 && DecNum <= 255)
      AskAgain = 0;
   else
   {
      AskAgain = 1;
      printf("\nYou entered a number “
            "not between 0 and 255\n\n");
   }
}
while (AskAgain);
```

```
do
{
   printf("Please enter a decimal "
          "number between 0 and 255 ");
   scanf("%d", &DecNum);
}
while (DecNum < 0 || DecNum > 255);
```

So which is better?

```
printf("Please enter a decimal "
       "number between 0 and 255 ");
scanf("%d", &DecNum);

while (DecNum < 0 || DecNum > 255)
{
   printf("\nYou entered a number "
          "not between 0 and 255\n\n");
   scanf("%d", &DecNum);
}
```

```
do
{
   printf("Please enter a decimal "
          "number between 0 and 255 ");
   scanf("%d", &DecNum);

}
while (DecNum < 0 || DecNum > 255);
```



```
while (not edge) {
    run();
}
```

```
do {
    run();
} while (not edge);
```

The switch Statement

multiway decision statement

```
switch (expression)
{
     case c1:
          statement;
          break;
     case c2:
          statement;
          break;
     .
     .
          ult:
          statement
}
```

Integer type for const_expr means it can be a char as well

expression

expression must have one of the integer types.

case const_expr: statement

const_expr must be a constant expression and must have one of the integer types.

There can be multiple case labeled statements but each const_expr must have distinct value.

default: statement

optional

Executed if none of the case statements are executed.

The `switch` Statement

```
if (MenuChoice == 1)
{
   printf("strlen() example\n");
}
else if (MenuChoice == 2)
{
   printf("strcpy() example\n");
}
else if (MenuChoice == 3)
{
   printf("strcat() example\n");
}
else
   printf("Invalid choice\n");
```

```
switch (MenuChoice)
{
   case 1:
       printf("strlen() example\n");
       break;
   case 2:
       printf("strcpy() example\n");
       break;
   case 3:
       printf("strcat() example\n");
       break;
   default:
       printf("Invalid menu choice\n");

}
```

The `switch` Statement

```
switch (MenuChoice)
{
   case 1:
      printf("strlen() example\n");
      break;
   case 2:
      printf("strcpy() example\n");
      break;
   case 3:
      printf("strcat() example\n");
      break;
   default:
      printf("Invalid menu choice\n");

}
```

```
switch (MenuChoice)
{
   case 1:
      printf("strlen() example\n");
   case 2:
      printf("strcpy() example\n");
   case 3:
      printf("strcat() example\n");
   default:
      printf("Invalid menu choice\n");
}
```

`casebreakDemo.c`

Altering the Flow of Control

`continue` and `break`

used to alter the flow of control

- `while` loop
- `for` loop
- `do-while` loop

`break` can also be used with `switch`

The `continue` Statement

```
continue;
```

- used inside a loop
- when encountered, it causes control to pass to the point after the last statement in the loop body instead of executing the next statement

The `break` Statement

```
break;
```

- used inside a loop or `switch`
- when encountered, it causes the loop to terminate and control to pass to the point immediately after the loop

```c
while (SecretNumber)
{
  printf("Enter a secret number between 1 and 10 : ");
  scanf("%d", &SecretNumber);

  if (SecretNumber < 1 || SecretNumber > 10)
  {
    printf("\n\nYou did not enter a number "
           "between 1 and 10.  Try again.\n\n\n");
    continue;
  }
  else
  {
    break;
  }
}

printf("\n\nPlayer 2\n\n");
```

```
Enter a secret number between 1 and 10 : 11

You did not enter a number between 1 and 10.  Try again.

Enter a secret number between 1 and 10 : 2

Player 2
```

whilebreakDemo.c

```c
do
{
  printf("Pick a number between 1 and 10 ");

  scanf("%d", &GuessedNumber);
  if (GuessedNumber < 1 || GuessedNumber > 10)
  {
    printf("\nYou did not enter a number between 1 and 10.  Try again.\n");
    continue;
  }

  if (GuessedNumber == SecretNumber)
  {
    printf("\n\nYou guessed the secret number!\n\n");
    break;
  }
  else
  {
    printf("\n\nThe number you entered is not the secret number.\n\n");
  }
}
while (GuessedNumber);

printf("\n\nBye!  Thanks for playing.\n");
```

whilebreakDemo.c

```
printf("Enter a secret number between 1 and 10 : ");
scanf("%d", &SecretNumber);

while (SecretNumber < 1 || SecretNumber > 10)
{
    printf("\n\nYou did not enter a number "
           "between 1 and 10.  Try again.\n\n\n"
           "Enter a secret number between 1 and 10 : ");
    scanf("%d", &SecretNumber);
}

printf("\n\nPlayer 2\n\n");
```

```
Enter a secret number between 1 and 10 : 11

You did not enter a number between 1 and 10.  Try again.

Enter a secret number between 1 and 10 : 2

Player 2
```

whilebreakDemo.c

```
printf("Do you want to print even or odd numbers (E/O) ");
scanf("%s", &EvenOdd);
printf("\n\nEnter start of range ");
scanf("%d", &Start);
printf("\n\nEnter end of range ");
scanf("%d", &End);
```

```
if (EvenOdd == 'O')
{
   for (i = Start; i <= 100; i++)
   {
      if (i > End)
         break;

      if (i & 1)
         printf("i = %d\n", i);
      else
         continue;
   }
}
```

```
else  /* assume E */
{
   for (i = Start; i <= 100; i++)
   {
      if (i > End)
         break;

      if (!(i & 1))
         printf("i = %d\n", i);
      else
         continue;
   }
}
```

forcontinue1Demo.c

```
if (EvenOdd == 'O')
{
   for (i = Start; i <= 100; i++)
   {
      if (i > End)
         break;

      if (i & 1)
         printf("i = %d\n", i);
      else
         continue;
   }
}
else  /* assume E */
{
   for (i = Start; i <= 100; i++)
   {
      if (i > End)
         break;

      if (!(i & 1))
         printf("i = %d\n", i);
      else
         continue;
   }
}
```

```
while (SecretNumber)
{
   printf("Enter a secret number … 1 and 10 : ");
   scanf("%d", &SecretNumber);

   if (SecretNumber < 1 || SecretNumber > 10)
   {
      printf("\n\nYou did not enter a number "
             "between 1 and 10.  Trygain.\n\n\n");
      continue;
   }
   else
   {
      break;
   }
}
```

```
while (SecretNumber < 1 || SecretNumber > 1
{
    printf("\n\nYou did not enter a number
           "between 1 and 10.  Try again.\
          "Enter a secret between 1 and 10
   scanf("%d", &SecretNumber);
}
```

```
while (!DISCREAD (ordfd,(short *)&gstOrdhdr,sizeof (gstOrdhdr)))
{
   /* Do not process invoices that are in process or marked as */
   /* duplicates (status 'D')                                  */
   if (gstOrdhdr.xinvoice == 'A' ||
       gstOrdhdr.xinvoice == 'B' ||
       gstOrdhdr.xinvoice == 'C' ||
       gstOrdhdr.xinvoice == 'D')
   {
       continue;
   }

   /* Order will be written to the transmit file so now add it to */
   /* the =srt_tbl in order to detect duplicates.  If it is a     */
   /* duplicate, then skip this order.                            */
   if (insert_dup_check())
   {
      continue;
   }
```

```
while (!DISCREAD (ordfd,(short *)&gstOrdhdr,sizeof (gstOrdhdr)))
{

   if (time_is_up())   /* it's time to finish up */
   {
      x = NBR_STATS;
      break;
   }


   if (nDone)          /* no room for order so quit*/
   {
      nDone = 0;
      break;
   }


}
```

```c
/* Based on value of x, set status. */
char get_stat (short x)
{
   switch  (x)
   {
      case 0:
         status = '5';
         break;
      case 1:
         status = 'L';
         break;
      case 2:
         status = 'T';
         break;
      case 3:
         status = '6';
         break;
      default:
         status = 'T';
         break;
   }
   return status;
}
```

The `return` Statement

```
return;
return expression;
```

- `return` statement is used in `main()` to terminate the program
- `return` statement can also be used to terminate execution of a function
- when `return` is executed, it causes control to pass from the function back to the position where it was called
- used to provide a point of exit from a function other than at the end of the function
- allows a function to return a value

```
/* Return TRUE if it's time to stop running.                               */

short time_is_up(void)
{
    TIME (time_tbl);

    if( !cutoff_hour )
       return FALSE;     /* Only one run */

    if( time_tbl[3] > cutoff_hour )
       return TRUE;

    if( (time_tbl[3] == cutoff_hour) && (time_tbl[4] > cutoff_minute))
       return TRUE;

    return FALSE;
}
```

```
/*   Don't go to delay if there is not enough time left to make another run   *
     after returning from delay.                                              */
short no_time_left(void)
{
    short future_hour = 0;
    short future_minute = 0;

    if( !cutoff_hour )  /* only one run */
       return TRUE;

    future_minute = (short)(( time_tbl[4] + delay_time) % 60);
    future_hour  = (short)(time_tbl[3] + (time_tbl[4] + delay_time)/ 60);
    if( future_hour > cutoff_hour)
       return TRUE;

    if( (future_hour == cutoff_hour) && (future_minute > cutoff_minute) )
       return TRUE;

    return FALSE;
}
```

The `exit()` Library Function

The `exit()` function takes a single parameter of type `int`.

- when executed, `exit()` causes the program to terminate
- control is returned to the operating system
- `<stdlib.h>` must be `include`d in your program

```
exit(0);
exit(1);
exit(255);
```

```
if ( nError = STARTOPENS((short *)"COMMENT ", &cmtfd,
                                                R_O+SHARED, 1, dataset) )
{
   sprintf(gszMsg, "opn_files() Error %d opening COMMENT file \n", nError);
   fnProcessError();
   SENDEMAIL((short *)&gstErrorEmail);
   msgabend (gszMsg, (short)nError, 0);
}
```

Library function with `exit(-1)` as its last statement

```
if ( nError = STARTOPENS((short *)"CUSTNAME", &cstfd,
                                                R_O+SHARED, 1, dataset) )
{
   sprintf(gszMsg, "opn_files() Error %d opening CUSTNAME file \n", nError);
   fnProcessError();
   SENDEMAIL((short *)&gstErrorEmail);
   msgabend (gszMsg, (short)nError, 0);
}
```

Library function with `exit(-1)` as its last statement

`return 0` vs `exit(0)` in `main()`

In `main()`,

what is the difference between using `return 0` and `exit(0)`?

```
int main(void)
{
   return 0;
}
```

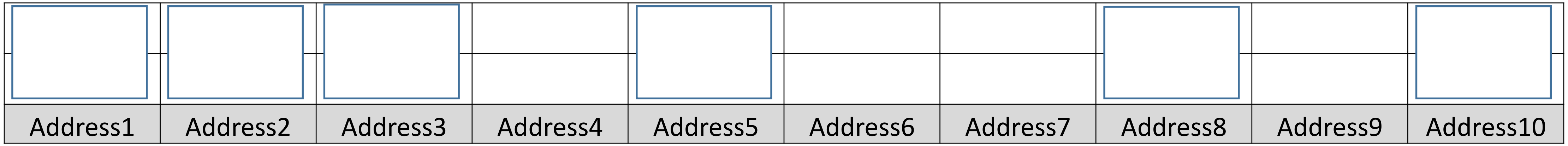
```
int main(void)
{
   exit(0);
}
```

`return` is a statement and `exit()` is a function. From a standard C perspective, there is no difference. There are, however, a few unusual circumstances where using `exit()` instead of `return` at the end of `main()` will cause undefined behavior; therefore, it is good practice to use `return` rather than `exit()` in `main()`.

Pointer Review

- Every variable has an address in memory

```
int VarA = 19;
int VarB = 32;
int VarC = 44;
int IntVar1 = 67
int IntVar2 = 23;
int IntVar3 = 66;
```



Address1	Address2	Address3	Address4	Address5	Address6	Address7	Address8	Address9	Address10

Pointer Review

- A pointer can hold that address

```
int *PtrVarA = &VarA;
int *PtrVarC = &VarC;
int *PtrIntVar1 = &IntVar1;
```

VarA	VarB	VarC		IntVar3			IntVar2		IntVar1
19	32	44		66			23		67
Address1	Address2	Address3	Address4	Address5	Address6	Address7	Address8	Address9	Address10

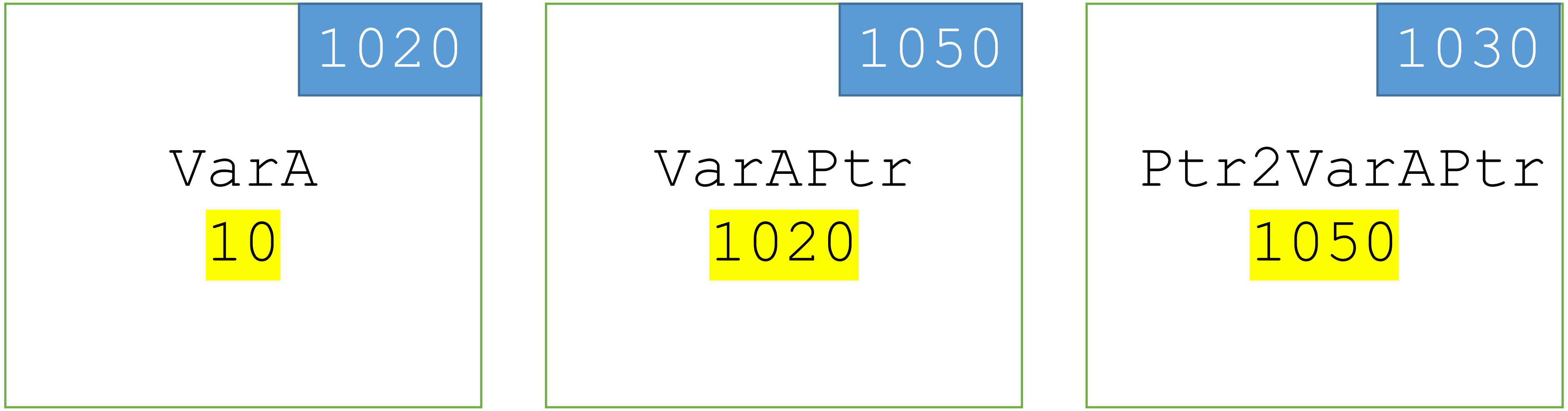
Pointer Review

- Dereferencing the pointer gets to the contents

```
printf(“Contents of PtrVarA %d”, *PtrVarA);
printf(“Contents of PtrVarC %d”, *PtrVarC);
printf(“Contents of PtrIntVar1 %d”, *PtrIntVar1);
```

VarA	VarB	VarC	PtrVarA	IntVar3	PtrVarC	PtrIntVar1	IntVar2		IntVar1
19	32	44	Address1	66	Address3	Address10	23		67
Address1	Address2	Address3	Address4	Address5	Address6	Address7	Address8	Address9	Address10

```
int VarA = 10;
int *VarAPtr = &VarA;
int **Ptr2VarAPtr = &VarAPtr;

printf("VarA = %d\n", VarA);
printf("*VarAPtr = %d\n", *VarAPtr);
printf("**Ptr2VarAPtr = %d\n", **Ptr2VarAPtr);
```



1020

VarA

10

1050

VarAPtr

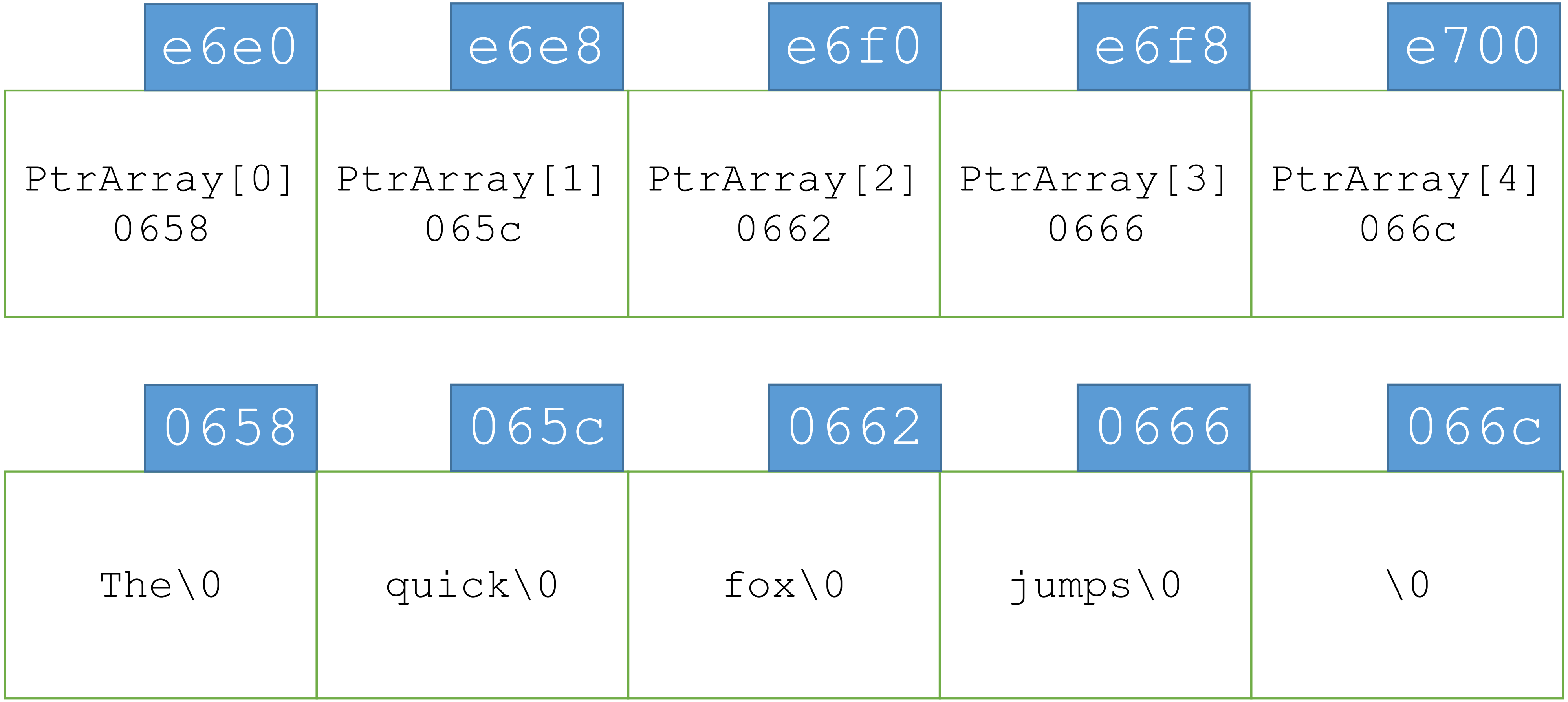
1020

1030

Ptr2VarAPtr

1050

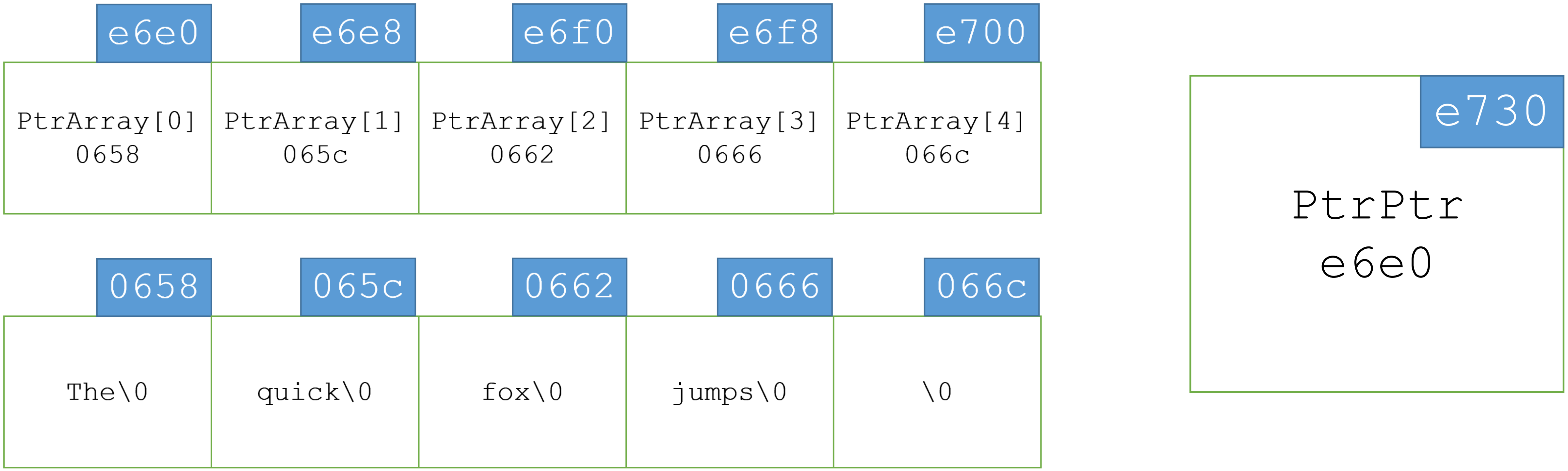
```c
char *PtrArray[] = {"The","quick","fox","jumps",""};
```



e6e0	e6e8	e6f0	e6f8	e700
PtrArray[0] 0658	PtrArray[1] 065c	PtrArray[2] 0662	PtrArray[3] 0666	PtrArray[4] 066c

0658	065c	0662	0666	066c
The\0	quick\0	fox\0	jumps\0	\0

```
char *PtrArray[] = {"The","quick","fox","jumps",""};
char **PtrPtr = PtrArray;
```



e6e0 PtrArray[0] 0658

e6e8 PtrArray[1] 065c

e6f0 PtrArray[2] 0662

e6f8 PtrArray[3] 0666

e700 PtrArray[4] 066c

0658 The\0

065c quick\0

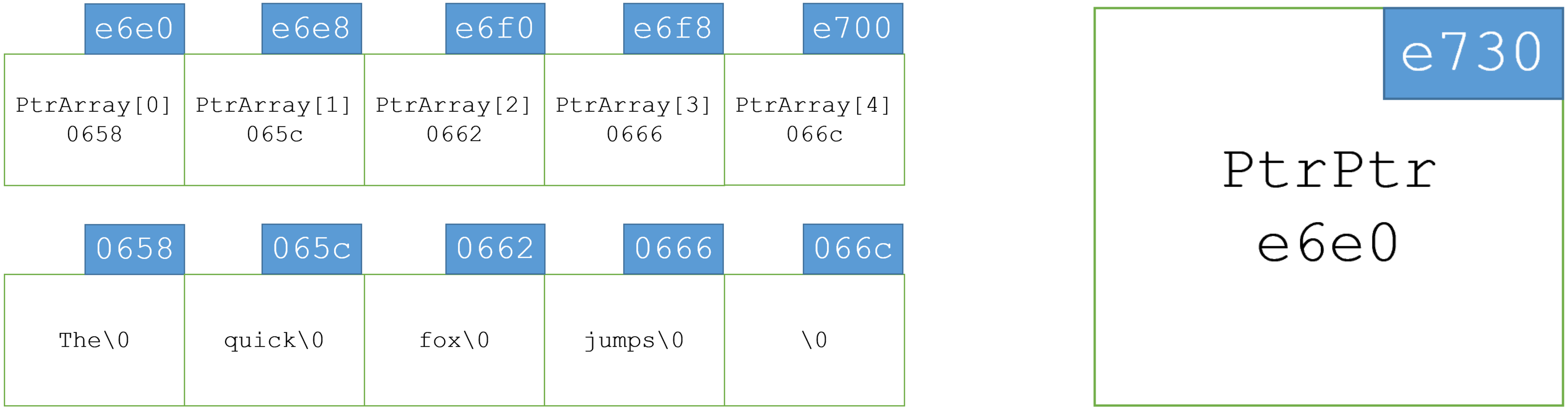
0662 fox\0

0666 jumps\0

066c \0

e730 PtrPtr e6e0

```
char *PtrArray[] = {"The","quick","fox","jumps",""};
char **PtrPtr = PtrArray;
```



e6e0	e6e8	e6f0	e6f8	e700
PtrArray[0] 0658	PtrArray[1] 065c	PtrArray[2] 0662	PtrArray[3] 0666	PtrArray[4] 066c

0658	065c	0662	0666	066c
The\0	quick\0	fox\0	jumps\0	\0

e730

PtrPtr
e6e0

```
for (i = 0; i < 5; i++)
{
   printf("PtrPtr + %d = %s\n", i, *(PtrPtr + i));
}
```

Arrays of Pointers

```
char *PtrArray[9];
```

`PtrArray` is of type `char *` so `PtrArray` is a pointer to `char`

`[9]` tells us that we have an array with 9 elements so

```
char *PtrArray[9]
```

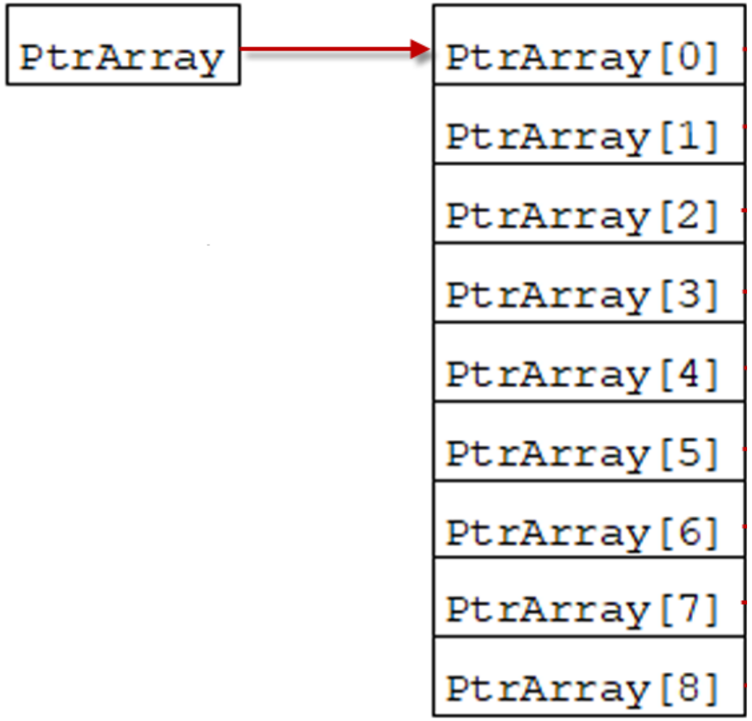
is an array of 9 pointers to `char`

Arrays of Pointers

```
char *PtrArray[9];
```

This construct is used in C to represent an array of strings.

The array name, `PtrArray`, is evaluated as the address of the first element of the array



PtrArray

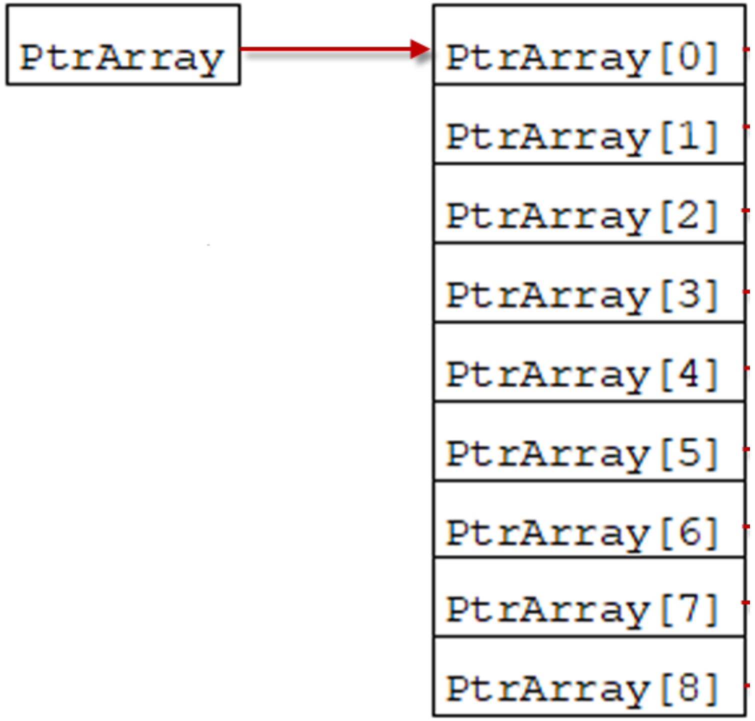
PtrArray[0]
PtrArray[1]
PtrArray[2]
PtrArray[3]
PtrArray[4]
PtrArray[5]
PtrArray[6]
PtrArray[7]
PtrArray[8]

Each element of `PtrArray` will be used to hold the address of a chunk of memory that contains a string.

Arrays of Pointers

```
char *PtrArray[8];
```

```
char *PtrArray[] = {"The","quick","fox","jumps","over","the","lazy","dog",""};
```



PtrArray

PtrArray[0]
PtrArray[1]
PtrArray[2]
PtrArray[3]
PtrArray[4]
PtrArray[5]
PtrArray[6]
PtrArray[7]
PtrArray[8]

```
PtrArray[0] = 0x400638
PtrArray[1] = 0x40063c
PtrArray[2] = 0x400642
PtrArray[3] = 0x400646
PtrArray[4] = 0x40064c
PtrArray[5] = 0x400651
PtrArray[6] = 0x400655
PtrArray[7] = 0x40065a
PtrArray[8] = 0x40065e
```

```
char *PtrArray[] = {"The","quick","fox","jumps","over","the","lazy","dog",""};

int i;

printf("sizeof(PtrArray) = %d\n"             rArray));

for (i = 0; i < 9; i++)
{
   printf("PtrArray[%d]) =          , PtrArray[i]);
   printf("PtrArray[%d])          i, *(PtrArray[i]));
}
```

```
sizeof(PtrA         = 72
```

What happens when we initialize a variable with a quoted string?

```
PtrArray[0]) = The
PtrArray[0]) = T
PtrArray[1]) = quick
PtrArray[1]) = q
PtrArray[2]) = fox
PtrArray[2]) = f
PtrArray[3]) = jumps
PtrArray[3]) = j
PtrArray[4]) = over
PtrArray[4]) = o
PtrArray[5]) = the
PtrArray[5]) = t
PtrArray[6]) = lazy
PtrArray[6]) = l
PtrArray[7]) = dog
PtrArray[7]) = d
PtrArray[8]) =
PtrArray[8]) =
```

ptr1arrayDemo.c

Double Indirection

Since any type in C can have a pointer to it, we can declare a pointer to a pointer.

```
int **ptr;
```

`*ptr` is a pointer to an `int` so `**ptr` is a pointer to a pointer to an `int`.

Double Indirection

```
char *PtrArray[] = {"The","quick","fox","jumps","over","the","lazy","dog",""};

char **PtrPtr = PtrArray;

int i;

printf("sizeof(PtrPtr)        %d\n"
       "sizeof(PtrArray)      %d\n",
        sizeof(PtrPtr), sizeof(PtrArray));

for (i = 0; i < 9; i++)
{
      printf("sizeof(PtrArray[%d]) = %d\n", i, sizeof(PtrArray[i]));
}
```

```
sizeof(PtrPtr)        8
sizeof(PtrArray)      72
```

```
sizeof(PtrArray[0]) = 8
sizeof(PtrArray[1]) = 8
sizeof(PtrArray[2]) = 8
sizeof(PtrArray[3]) = 8
sizeof(PtrArray[4]) = 8
sizeof(PtrArray[5]) = 8
sizeof(PtrArray[6]) = 8
sizeof(PtrArray[7]) = 8
sizeof(PtrArray[8]) = 8
```

dichar1Demo.c

dichar2Demo.c

Double Indirection

```
char *PtrArray[] = {"The","quick","fox","jumps","over","the","lazy","dog",""};

char **PtrPtr = PtrArray;

int i;

for (i = 0; i < 9; i++)
{
   printf("PtrArray[%d] = %s\n",
           i, PtrArray[i]);
}

for (i = 0; i < 9; i++)
{
   printf("PtrPtr + %d = %s\n",
           i, *(PtrPtr + i));
}
```

```
PtrArray[0] = The
PtrArray[1] = quick
PtrArray[2] = fox
PtrArray[3] = jumps
PtrArray[4] = over
PtrArray[5] = the
PtrArray[6] = lazy
PtrArray[7] = dog
PtrArray[8] =
PtrPtr + 0 = The
PtrPtr + 1 = quick
PtrPtr + 2 = fox
PtrPtr + 3 = jumps
PtrPtr + 4 = over
PtrPtr + 5 = the
PtrPtr + 6 = lazy
PtrPtr + 7 = dog
PtrPtr + 8 =
```

dichar3Demo.c

Double Indirection

```
char *PtrArray[] = {"The","quick","fox","jumps","over","the","lazy","dog",""};

char **PtrPtr = PtrArray;

int i = 0;

while (PtrArray[i] != "")
{
   printf("PtrArray[%d] = %s\n", i++, PtrArray[i]);
}

i = 0;
while (*(PtrPtr + i) != "")
{
   printf("PtrPtr + %d = %s\n", i++, *(PtrPtr + i));
}
```

```
PtrArray[0] = The
PtrArray[1] = quick
PtrArray[2] = fox
PtrArray[3] = jumps
PtrArray[4] = over
PtrArray[5] = the
PtrArray[6] = lazy
PtrArray[7] = dog
PtrPtr + 0 = The
PtrPtr + 1 = quick
PtrPtr + 2 = fox
PtrPtr + 3 = jumps
PtrPtr + 4 = over
PtrPtr + 5 = the
PtrPtr + 6 = lazy
PtrPtr + 7 = dog
```

dichar4Demo.c

Double Indirection

```
char *PtrArray[] = {"The","quick","fox","jumps","over","the","lazy","dog",""};

char **PtrPtr = PtrArray;

int i = 0;

while (PtrArray[i] != "")
{
    printf("*PtrArray[%d] = %c\n", i++, *PtrArray[i]);
}

i = 0;
while (*(PtrPtr + i) != "")
{
    printf("**(PtrPtr + %d) = %c\n", i++, **(PtrPtr + i));
}
```

```
%s    PtrArray[i]
```

```
%s    *(PtrPtr + i)
```

```
*PtrArray[0] = T
*PtrArray[1] = q
*PtrArray[2] = f
*PtrArray[3] = j
*PtrArray[4] = o
*PtrArray[5] = t
*PtrArray[6] = l
*PtrArray[7] = d
**(PtrPtr + 0) = T
**(PtrPtr + 1) = q
**(PtrPtr + 2) = f
**(PtrPtr + 3) = j
**(PtrPtr + 4) = o
**(PtrPtr + 5) = t
**(PtrPtr + 6) = l
**(PtrPtr + 7) = d
```

dichar5Demo.c

Double Indirection

```
char *PtrArray[] = {"The","quick","fox","jumps","over","the","lazy","dog",""};
char **PtrPtr = PtrArray;

while (PtrArray[i] != "")
{
   for (j = 0; j < strlen(PtrArray[i]); j++)
   {
      printf("PtrArray[%d][%d] = %c\n",
               i, j, PtrArray[i][j]);
   }
   i++;
}
```

```
PtrArray[0][0] = T   PtrArray[4][0] = o
PtrArray[0][1] = h   PtrArray[4][1] = v
PtrArray[0][2] = e   PtrArray[4][2] = e
PtrArray[1][0] = q   PtrArray[4][3] = r
PtrArray[1][1] = u   PtrArray[5][0] = t
PtrArray[1][2] = i   PtrArray[5][1] = h
PtrArray[1][3] = c   PtrArray[5][2] = e
PtrArray[1][4] = k   PtrArray[6][0] = l
PtrArray[2][0] = f   PtrArray[6][1] = a
PtrArray[2][1] = o   PtrArray[6][2] = z
PtrArray[2][2] = x   PtrArray[6][3] = y
PtrArray[3][0] = j   PtrArray[7][0] = d
PtrArray[3][1] = u   PtrArray[7][1] = o
PtrArray[3][2] = m   PtrArray[7][2] = g
PtrArray[3][3] = p
PtrArray[3][4] = s
```

dichar5Demo.c

Double Indirection

```
char *PtrArray[] = {"The","quick","fox","jumps","over","the","lazy","dog",""};
char **PtrPtr = PtrArray;

i = 0;
while (*(PtrPtr + i) != "")
{
   for (j = 0; j < strlen(*(PtrPtr + i)); j++)
   {
      printf("*(*(PtrPtr + %d) + %d) = %c\n",
                i, j, *(*(PtrPtr + i)+j));
   }
   i++;
}
```

```
*(*(PtrPtr + 4) + 0) = o
*(*(PtrPtr + 4) + 1) = v
*(*(PtrPtr + 4) + 2) = e
*(*(PtrPtr + 4) + 3) = r
*(*(PtrPtr + 5) + 0) = t
*(*(PtrPtr + 5) + 1) = h
*(*(PtrPtr + 5) + 2) = e
*(*(PtrPtr + 6) + 0) = l
*(*(PtrPtr + 6) + 1) = a
*(*(PtrPtr + 6) + 2) = z
*(*(PtrPtr + 6) + 3) = y
*(*(PtrPtr + 7) + 0) = d
*(*(PtrPtr + 7) + 1) = o
*(*(PtrPtr + 7) + 2) = g
```

Double Indirection

C does not put a limit on the number of levels of indirection.

```
long ***ThisIsRidiculous;
```

Pointer to a pointer to a pointer to a `long`

```
char *****ThisIsMoreRidiculous;
```

Pointer to a pointer to a pointer to a pointer to a pointer to a `char` diDemo.c

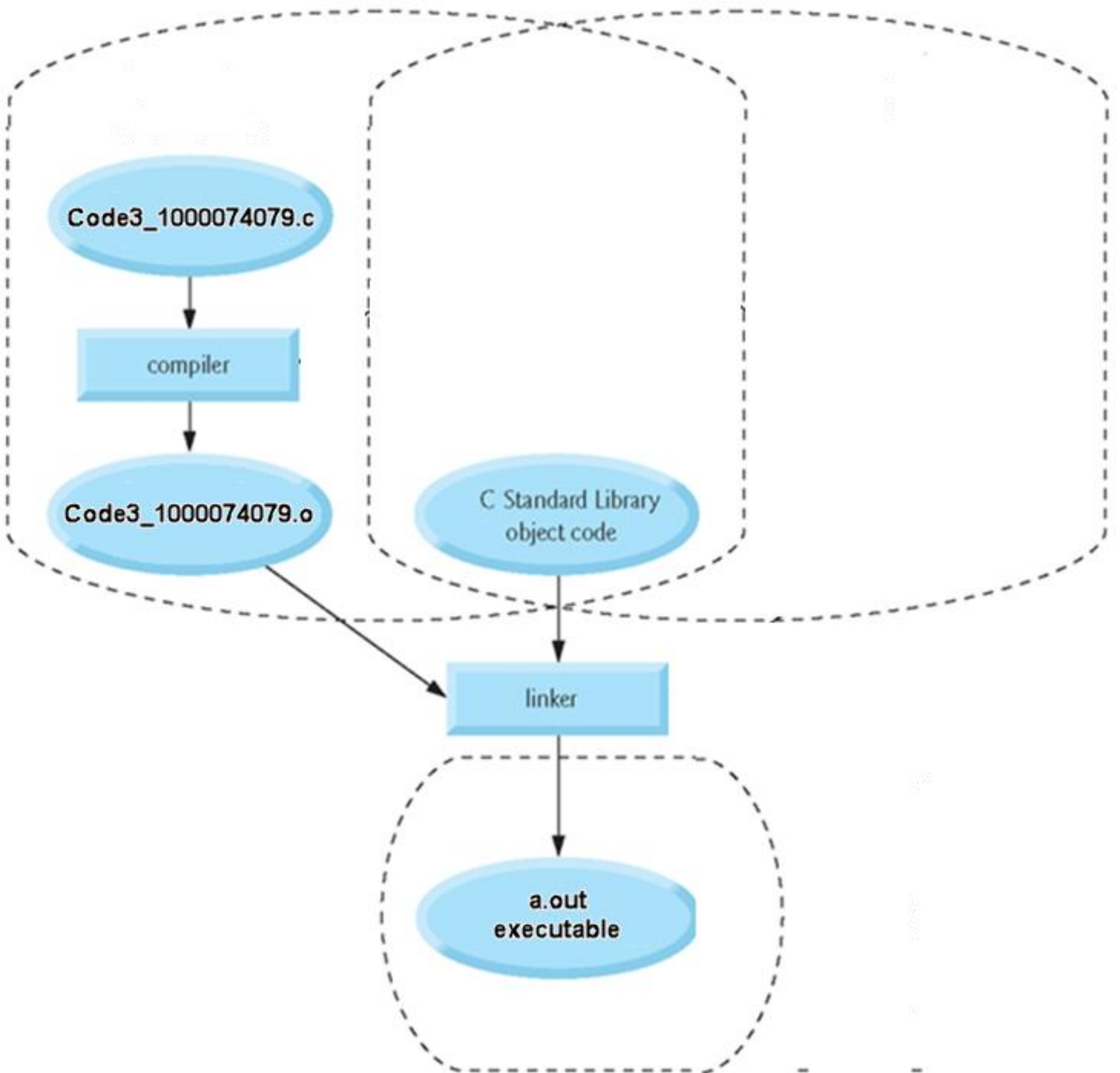
Double Indirection

```
#include <stdio.h>

int main(void)
{
  int VarA = 10;
  int *VarAPtr = &VarA;
  int **Ptr2VarAPtr = &VarAPtr;

  printf("VarA = %d\n", VarA);
  printf("*VarAPtr = %d\n", *VarAPtr);
  printf("**Ptr2VarAPtr = %d\n", **Ptr2VarAPtr);

  return 0;
}
```

```
(gdb) p VarA
$1 = 10
(gdb) p &VarA
$2 = (int *) 0x7fffffffe7a4
(gdb) p VarAPtr
$3 = (int *) 0x7fffffffe7a4
(gdb) p &VarAPtr
$4 = (int **) 0x7fffffffe798
(gdb) p Ptr2VarAPtr
$5 = (int **) 0x7fffffffe798
```

```
VarA = 10
*VarAPtr = 10
**Ptr2VarAPtr = 10
```

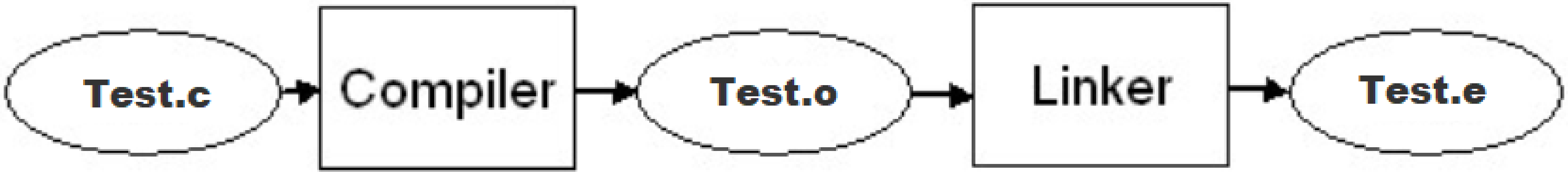
```
gcc Code3_1000074079.c
```



makefile

```
gcc -c Test.c
```

```
gcc -g Test.o -o Test.e
```



The source file that you type into the editor. This is just a text file, anybody can read.

The object file is an intermediate file. It is only readable by the compiler and the linker.

The executable is the final product. It is a binary file that the operating system can run.

```
[frenchdm@omega CA1]$ gcc -c Code1_1000074079.c

[frenchdm@omega CA1]$ ls
Code1_1000074079.c  Code1_1000074079.o

[frenchdm@omega CA1]$ gcc -g Code1_1000074079.o -o Code1_1000074079.e

[frenchdm@omega CA1]$ ls
Code1_1000074079.c  Code1_1000074079.e  Code1_1000074079.o

[frenchdm@omega CA1]$ Code1_1000074079.e
Decimal to binary convertor

Please enter a decimal number between 0 and 255 170

Decimal 170 converts to binary 10101010
[frenchdm@omega CA1]$
```

`makefile`

What is a `makefile`?

`make` is UNIX utility that is designed to start execution of a `makefile`.

A `makefile` is a special file, containing shell commands, that you create and name `makefile`.

While in the directory containing your `makefile`, you will type `make` and the commands in the `makefile` will be executed.

If you create more than one `makefile`, be certain you are in the correct directory before typing `make`.

`makefile`

`make` keeps track of the last time files (normally object files) were updated and only updates those files which are required (ones containing changes) to keep the sourcefile up-to-date.

If you have a large program with many source and/or header files, when you change a file on which others depend, you must recompile all the dependent files.

Without a `makefile`, this is an extremely time-consuming task.

`makefile`

As a `makefile` is a list of shell commands, it must be written for the shell which will process the `makefile`. A `makefile` that works well in one shell may not execute properly in another shell.

The `makefile` contains a list of rules. These rules tell the system what commands you want to be executed. Most times, these rules are commands to compile(or recompile) a series of files.

The rules, **which must begin in column 1**, are in two parts. The first line is called a dependency line and the subsequent line(s) are system commands or recipes which must be indented with a tab.

`makefile`

After the `makefile` has been created, a program can be (re)compiled by typing `make` in the correct directory.

`make` then reads the `makefile` and creates a dependency tree and takes whatever action is necessary. It will not necessarily do all the rules in the `makefile` as all dependencies may not need updated. It will rebuild target files if they are missing or older than the dependency files.

Unless directed otherwise, `make` will stop when it encounters an error during the construction process.

makefile

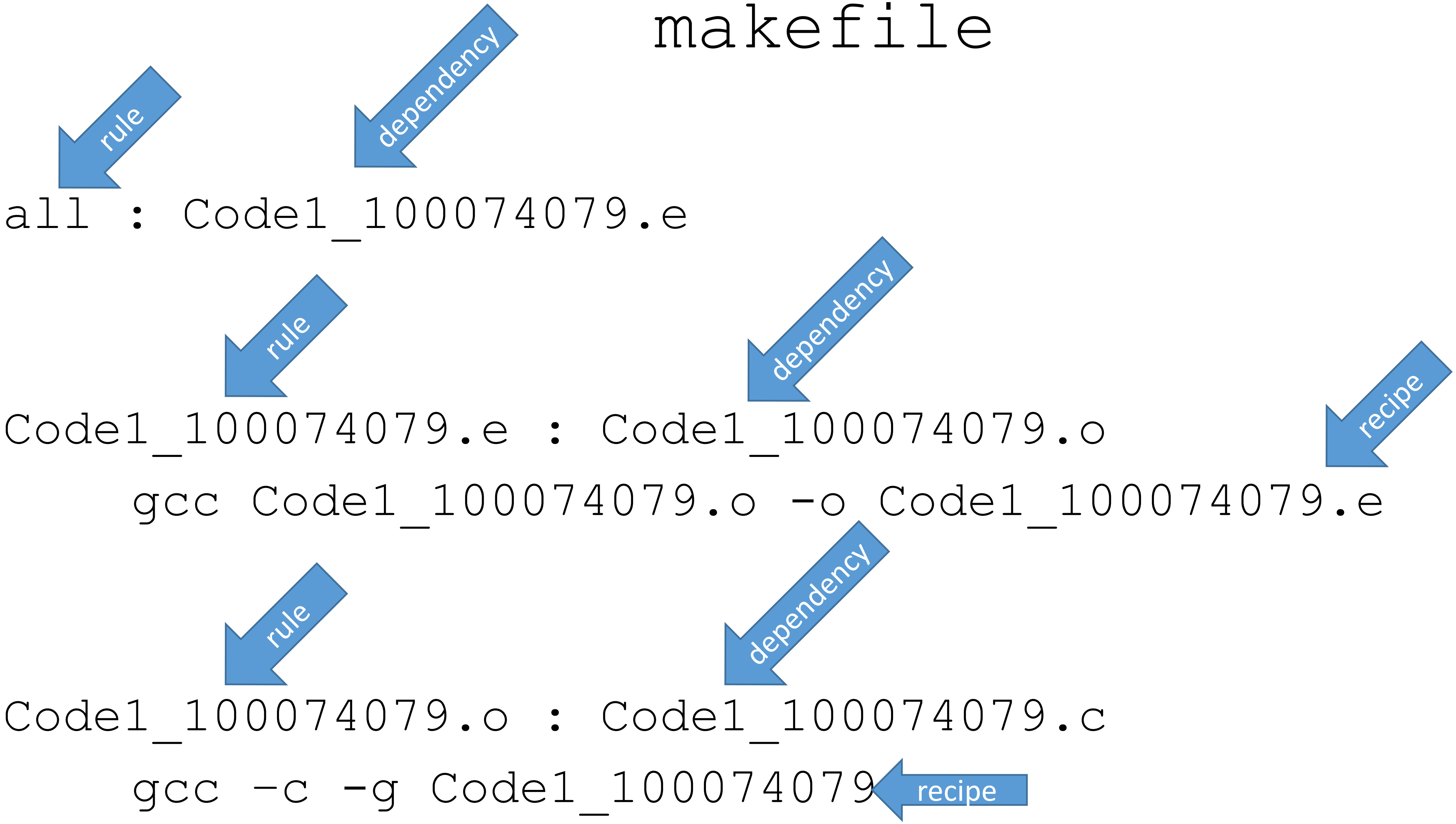
```
RULE : DEPENDENCIES
[tab]SYSTEM COMMANDS (RECIPE)
```

A **rule** is usually the name of a file that is generated by a program; examples of rules are executable or object files. A rule can also be the name of an action to carry out, such as "clean". Multiple rules must be separated by a space

A **dependency** (also called *prerequisite*) is a file that is used as input to create the rule. A rule often depends on several files.

The **system command(s)** (also called *recipe*) is an action that make carries out. A recipe may have more than one command, either on the same line or each on its own line. Recipe lines must be indented using a single <tab> character.

makefile

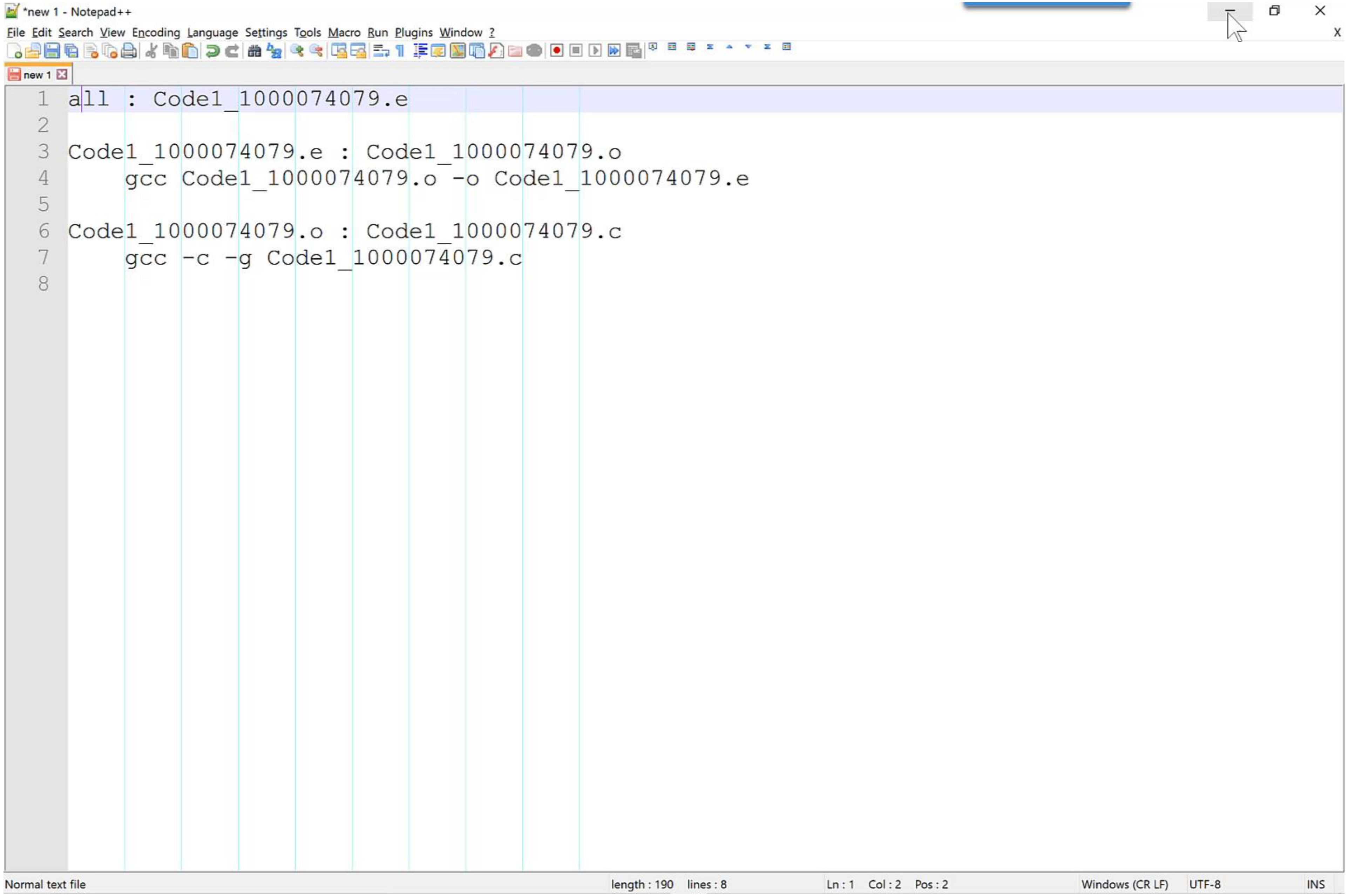


```
all : Code1_100074079.e

Code1_100074079.e : Code1_100074079.o
      gcc Code1_100074079.o -o Code1_100074079.e

Code1_100074079.o : Code1_100074079.c
      gcc -c -g Code1_100074079
```

makefile

```
all : Code1_100074079.e


Code1_100074079.e : Code1_100074079.o
     gcc Code1_100074079.o -o Code1_100074079.e


Code1_100074079.o : Code1_100074079.c
     gcc –c -g Code1_100074079.c
```

```
[frenchdm@omega CA1]$ more makefile
all : Code1_1000074079.e

Code1_1000074079.e : Code1_1000074079.o
        gcc Code1_1000074079.o -o Code1_1000074079.e

Code1_1000074079.o : Code1_1000074079.c
        gcc –c -g Code1_1000074079.c



[frenchdm@omega CA1]$ make
gcc –c -g Code1_1000074079.c
gcc Code1_1000074079.o -o Code1_1000074079.e

[frenchdm@omega CA1]$ ls
Code1_1000074079.c  Code1_1000074079.e  Code1_1000074079.o
makefile
```



```
all : Code1_1000074079.e

Code1_1000074079.e : Code1_1000074079.o
	gcc Code1_1000074079.o -o Code1_1000074079.e

Code1_1000074079.o : Code1_1000074079.c
	gcc -c -g Code1_1000074079.c

```

```
make

makefile:4: *** missing separator.  Stop.
```

If you get this `makefile` error,

then check that all of your recipes are indented with

TABS

and **not** with

SPACES.

```
[frenchdm@omega CA1]$ ls
Code1_1000074079.c  makefile.txt
[frenchdm@omega CA1]$ make
make: *** No targets specified and no makefile found.  Stop.

[frenchdm@omega CA1]$ mv makefile.txt makefile.mak
[frenchdm@omega CA1]$ ls
Code1_1000074079.c  makefile.mak

[frenchdm@omega CA1]$ make
make: *** No targets specified and no makefile found.  Stop.

[frenchdm@omega CA1]$ mv makefile.mak makefile
[frenchdm@omega CA1]$ make
gcc -c Code1_1000074079.c
gcc -g Code1_1000074079.o -o Code1_1000074079.e
[frenchdm@omega CA1]$
```

makefile

```
all : HelloWorld.e

HelloWorld.e : HelloWorld.o
      gcc HelloWorld.o -o HelloWorld.e

HelloWorld.o : HelloWorld.c
      gcc -c -g HelloWorld.c
```

With this explicit `makefile`, calling just “`make`” causes execution to start at rule `all`

Calling “`make HelloWorld.e`” causes execution to start at rule `HelloWorld.e`

Calling “`make HelloWorld.o`” causes execution to start at rule `HelloWorld.o`

makefile

```
SRC = Code2_100074079.c
OBJ = $(SRC:.c=.o)
EXE = $(SRC:.c=.e)

CFLAGS = -g

all : $(EXE)

$(EXE): $(OBJ)
	gcc $(OBJ) -o $(EXE)

$(OBJ) : $(SRC)
	gcc -c $(CFLAGS) $(SRC)
```

```
all : Code1_1000074079.e

Code1_1000074079.e : Code1_1000074079.o
	gcc Code1_1000074079.o -o Code1_1000074079.e

Code1_1000074079.o : Code1_1000074079.c
	gcc -c -g Code1_1000074079.c
```

makefile

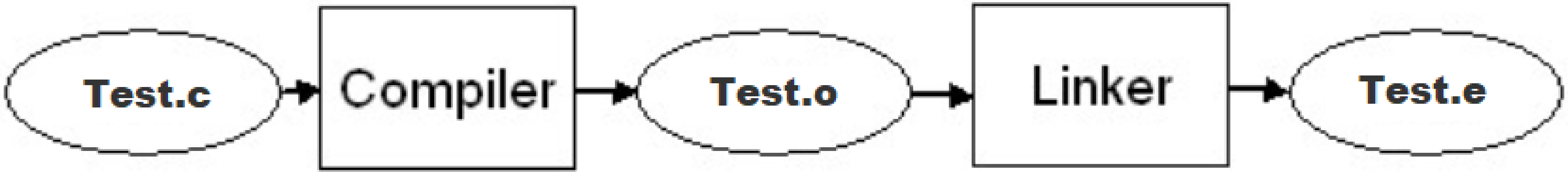
```
SRC = Test.c
OBJ = Test.o
EXE = Test.e

CFLAGS = -g

all :  Test.e


 Test.e  Test.o


      gcc Test.o  -o  Test.e


 Test.o  :  Test.c
      gcc -c        -g      Test.c
```

```
all : Test.e

Test.e : Test.o
           gcc Test.o -o Test.e

Test.o : Test.c
           gcc -c -g Test.c
```

makefile

```
gcc -c -g Test.c
```

```
gcc Test.o MyLib.o -o Test.e
```



The source file that you type into the editor. This is just a text file, anybody can read.

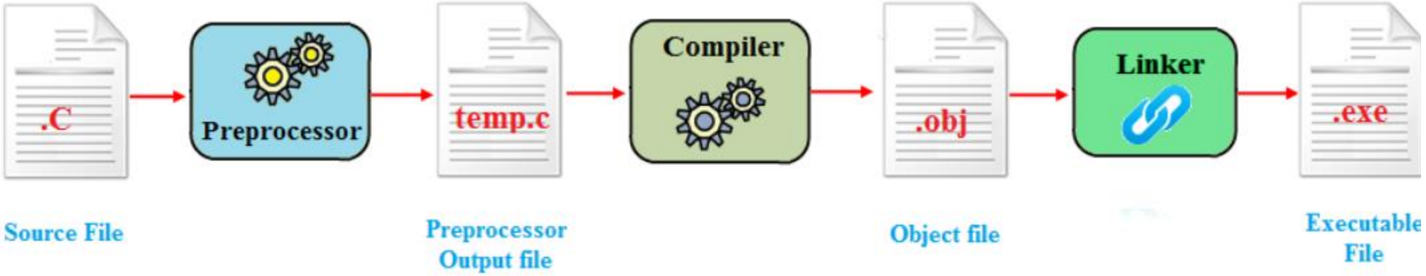
The object file is an intermediate file. It is only readable by the compiler and the linker.

The executable is the final product. It is a binary file that the operating system can run.

makefile

```
SRC1 = Code2_1000074079.c
SRC2 = DrawTool.c
OBJ1 = $(SRC1:.c=.o)
OBJ2 = $(SRC2:.c=.o)
EXE = $(SRC1:.c=.e)




CFLAGS = -g

all : $(EXE)

$(EXE): $(OBJ1) $(OBJ2)
      gcc $(OBJ1) $(OBJ2) -o $(EXE)

$(OBJ1) : $(SRC1)
      gcc -c $(CFLAGS) $(SRC1)

$(OBJ2) : $(SRC2)
      gcc -c $(CFLAGS) $(SRC2)
```

```
SRC = Code2_100074079.c
OBJ = $(SRC:.c=.o)
EXE = $(SRC:.c=.e)

CFLAGS = -g

all : $(EXE)

$(EXE): $(OBJ)
      gcc $(OBJ) -o $(EXE)

$(OBJ) : $(SRC)
      gcc -c $(CFLAGS) $(SRC)
```



compiler

creates an object file

linker

takes in object files and produces an executable file

```
SRC1 = Code2_1000074079.c
SRC2 = DrawTool.c
OBJ1 = $(SRC1:.c=.o)
OBJ2 = $(SRC2:.c=.o)
EXE = $(SRC1:.c=.e)



CFLAGS = -g

all : $(EXE)

$(EXE): $(OBJ1) $(OBJ2)
        gcc $(OBJ1) $(OBJ2) -o $(EXE)

$(OBJ1)  : $(SRC1)
        gcc -c $(CFLAGS) $(SRC1)

$(OBJ2)  : $(SRC2)
        gcc -c $(CFLAGS) $(SRC2)
```

```
SRC1 = Code2_1000074079.c
SRC2 = DrawTool.c
OBJ1 = $(SRC1:.c=.o)
OBJ2 = $(SRC2:.c=.o)
EXE = $(SRC1:.c=.e)
CFLAGS = -g
all : $(EXE)
$(EXE): $(OBJ1) $(OBJ2)
	gcc $(OBJ1) $(OBJ2) -o $(EXE)
$(OBJ1) : $(SRC1)
	gcc -c $(CFLAGS) $(SRC1)
$(OBJ2) : $(SRC2)
	gcc -c $(CFLAGS) $(SRC2)
```

Library

Libraries are not executable

- do not contain a `main()` function
- only contain functions and declarations
- you cannot `gcc` a library and then `./a.out` it

Library

Your library will consist of two files

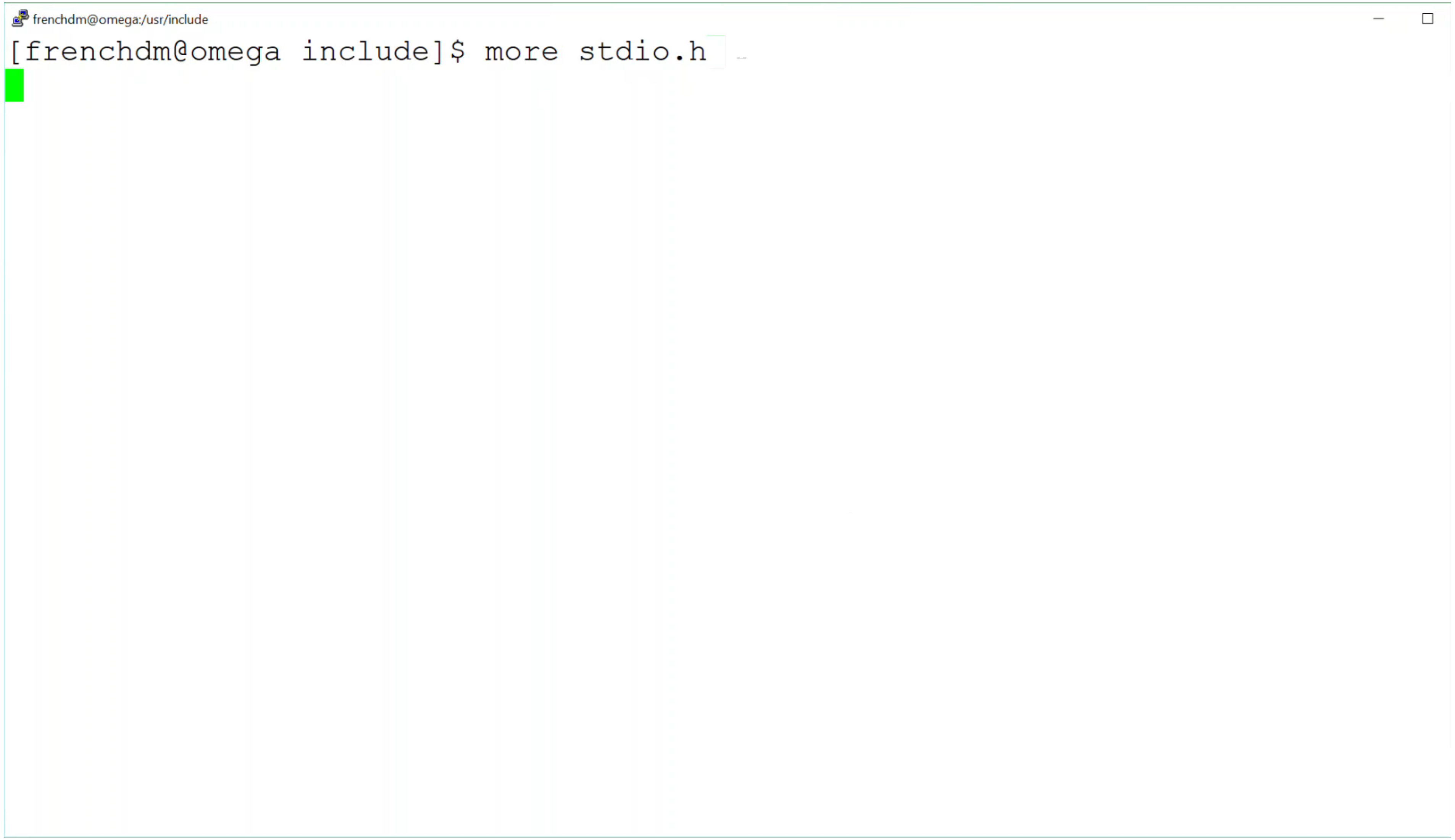
`MyLibray.c`

C file containing your library functions and code

`MyLibrary.h`

Header file containing the function prototypes for your library

stdio.h



```
[frenchdm@omega include]$ more stdio.h
```

Creating a library

`MyLibrary.h`

Create `MyLibrary.h` and move prototypes from `Code.c` to `MyLibrary.h`

Add `include` guard

`MyLibrary.c`

Create `MyLibrary.c` and move `Fun1()` and `Fun2()` code from `Code.c` to `MyLibrary.c`

Add includes

`stdio.h`

`MyLibrary.h`

`Code.c`

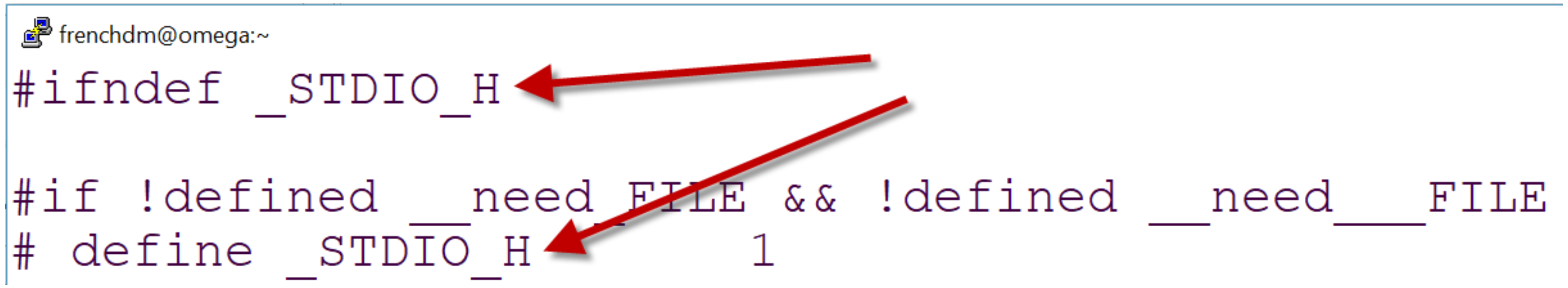
Add include for `MyLibrary.h`

`makefile`

Create a `makefile` that compiles/links two object files.

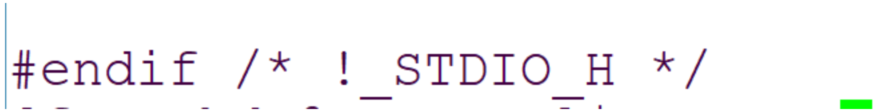
Special Note about Your Header Files

If you look at any system include you will see



```
#ifndef _STDIO_H

#if !defined __need_FILE && !defined __need___FILE
# define _STDIO_H       1
```

at the beginning and



```
#endif /* !_STDIO_H */
```

at the end

Special Note about Your Header Files

In the **C** and **C++** programming languages, an #include **guard**, sometimes called a macro **guard** or **header guard**, is a particular construct used to avoid the problem of double inclusion when dealing with the include directive. The addition of #include **guards** to a **header** file is one way to make that file idempotent.

Idempotence is the property of certain operations in mathematics and computer science whereby they can be applied multiple times without changing the result beyond the initial application.

Special Note about Your Header Files

Add `#include` **guard to** `MyLibrary.h`

```
#ifndef _MYLIBRARY_H
#define _MYLIBRARY_H

void Fun1(int []);
void Fun2(int []);

#endif
```

Include guard

Given these functions, create the header file.

```
void myFunA(int x)
{
   printf("%d", x);
}

int myFunB(void)
{
   return 100;
}
```

Create a file named `TooFun.h`

```
#ifndef _TOOFUN_H
#define _TOOFUN_H

void myFunA(int x);
int myFunB(void);

#endif
```

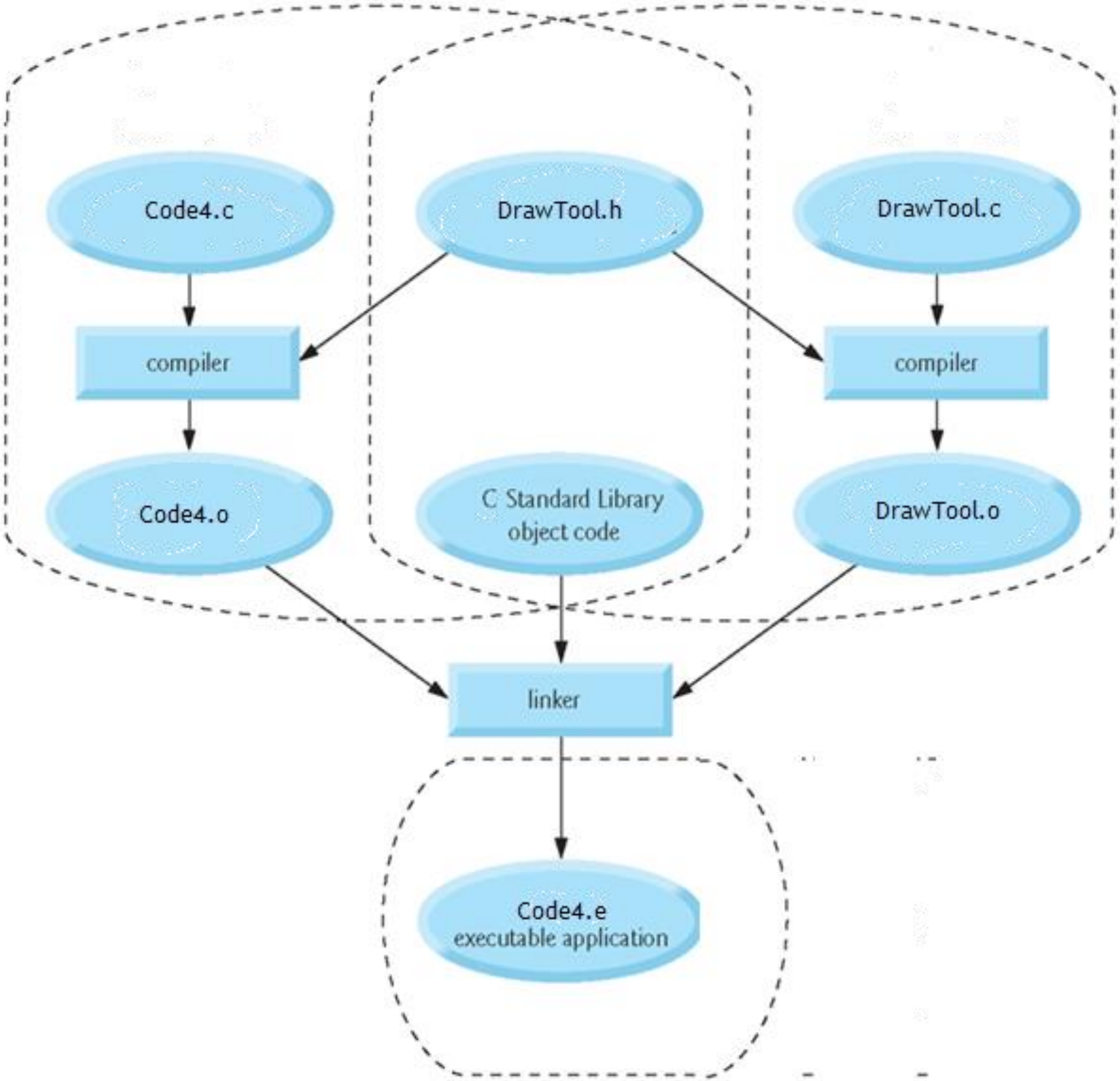
Include guard

Given these functions, create the header file.

```
void myFunA(int x[])
{
   printf("%d", x[0]);
}

void myFunB(int x[][4])
{
   printf("%d", x[0][0];
}
```

Create a file named `NoFun.h`

```
#ifndef _BUNNY_H
#define _BUNNY_H

void myFunA(int);
void myFunB(int[][4]);

#endif
```


Code4.c
DrawTool.h
DrawTool.c
compiler
compiler
Code4.o
C Standard Library object code
DrawTool.o
linker
Code4.e executable application

Compiling and Linking

source files for one executable

Module A

Contains main()

Prompts user for input

Call various functions based on input

Module B

Functions to open files

Functions to read files

Functions to write to files

Module C

Functions to perform FTP actions

an object file is created for each module and then the linker puts the objects together to create an executable

```
SRC1 = Code2_1000074079.c

SRC2 = MyLibrary.c

OBJ1 = $(SRC1:.c=.o)

OBJ2 = $(SRC2:.c=.o)

EXE = $(SRC1:.c=.e)



CFLAGS = -g

all : $(EXE)

$(EXE): $(OBJ1) $(OBJ2)
       gcc $(CFLAGS) $(OBJ1) $(OBJ2) -o $(EXE)

$(OBJ1) : $(SRC1)
       gcc -c $(CFLAGS) $(SRC1) -o $(OBJ1)

$(OBJ2) : $(SRC2)
       gcc -c $(CFLAGS) $(SRC2) -o $(OBJ2)
```

Module A – `Code2_1000074079.c`

Module B – `MyLibrary.c`

Link both object files together to make an executable

Generate object file for `Code2_1000074079.c`

Generate object file for `MyLibrary.c`

```
# Donna French 1000074079
SRC1 = Code6_1000074079.c
SRC2 = QueueLib.c
SRC3 = BSTLib.c
SRC4 = ListLib.c
SRC5 = StackLib.c
OBJ1 = $(SRC1:.c=.o)
OBJ2 = $(SRC2:.c=.o)
OBJ3 = $(SRC3:.c=.o)
OBJ4 = $(SRC4:.c=.o)
OBJ5 = $(SRC5:.c=.o)
EXE = $(SRC1:.c=.e)

CFLAGS = -g

all : $(EXE)

$(EXE): $(OBJ1) $(OBJ2) $(OBJ3) $(OBJ4) $(OBJ5)
        gcc $(CFLAGS) $(OBJ1) $(OBJ2) $(OBJ3) $(OBJ4) $(OBJ5) -o $(EXE)

$(OBJ1) : $(SRC1)
        gcc -c $(CFLAGS) $(SRC1) -o $(OBJ1)

$(OBJ2) : $(SRC2)
        gcc -c $(CFLAGS) $(SRC2) -o $(OBJ2)

$(OBJ3) : $(SRC3)
        gcc -c $(CFLAGS) $(SRC3) -o $(OBJ3)

$(OBJ4) : $(SRC4)
        gcc -c $(CFLAGS) $(SRC4) -o $(OBJ4)

$(OBJ5) : $(SRC5)
        gcc -c $(CFLAGS) $(SRC5) -o $(OBJ5)
```

Note : comments can be added to a makefile using a # in the first column.

```
make -B
```