Day 20 Working with Memory

Type Conversions

Automatic Type Conversions

often referred to as implicit conversions

when C expression evaluated, if all components in the expression have the same type, resulting type is that type as well

```
int x,y;
z= x+y;
```

z will be int

least comprehensive to most comprehensive (like order of data types)

```
char
short
int
long
long long
float
double
long double
```

so ex: expression with int and char evaluates to type int

kinda like an order of operations to determine data type after evaluation

so: Y + X * 2

will determine data type of x*2 THEN determine data type of Y + (X*2)

within expressions operands can be **promoted** in pairs for each binary operator in the expression

following these rules:

```
1. if either operand is a long double, the other is promoted to type long double
2. if either is a double, the other is promoted to double
3. if either is a float, other promoted to float
4. either a long, other promoted to long
```

just makes a copy will not change underlying data type

Conversion by Assignment

assignment operator =

expression on right side always promoted to type of data object on left side

could cause a “demotion” rather than a “promotion”

Explicit conversions with typecasts

consists of a type name, in paranthesis before an expression. can be performed on arithmetic expressions and pointers

```
(float)i
```

most common use for arithmetic is to avoid losing fractional part of answer in integer division

```
f1 = (float)i1/i2;
```

Allocating Memory Storage Space

static memory allocation- like array

dynamic memory allocation- allocating memory storage at runtime

requires stdlib.h some compilers malloc.h

all allocation functions return type void pointer

every program needs a way to check to ensure memory was allocated correctly and means to gracefully exit if not

malloc()

can allocate storage for any storage need

```
void *malloc(size_t num);
```

num is number of bytes to allocate and returns pointer to first byte

```
int *ptr;
ptr = malloc(sizeof(int));
```

calloc()

```
void *calloc(size_t num, size_t size);
```

num number objects to allocate, size size in bytes of each object. returns pointer to first byte

realloc()

changes size of block of memory previously allocated with malloc() or calloc()

```
void *realloc(void *ptr, size_t size);
```

ptr to original block of memory. new size specified in bytes

outcomes: 1. if sufficient space memory allocated and returns ptr to adjusted block 2. if space does not exist, new block for size is allocated and exisitng data copied from old block to new, old block freed, returns poitner to new block 3. if ptr is null, acts like malloc() ie allocating a block of size bytes and returning pointer to it 4. argument size is 0, memory ptr points to is freed, and function returns to null 5. if memory insufficient for reallocation, function returns null and original block is unchanged

free()

```
void free(void *ptr);
```

Manipulating Memory Blocks

memset()- set all bytes in a block of memory to a particular value

```
void *memset(void *dest, int c, size_t count);
```

c is value to set, count is number of bytes, starting at dest, to be set. could do something like ex: changing array[50] do array+5 to change starting at 5th index

c range 0 to 255

memcpy()- copies blocks of data between memory blocks- does not care about data type

```
void *memcpy(void *dest, void *src, size_t count);
```

dest and src point to destination and source memory blocks. count specifies number bytes to be copied. dest return value.

does not handle overlapping memory blocks, therefore **should just use**

memmove()- same as memcpy() just handles overlapping memory blocks better

Bits

C bitwise operators let you manipulate individual bits of integer variables

shift operators- shift bits in integer variable by specified number of positions using

<< shift to left

shift to right

```
x << n
```

shifts bits in x n positions to the left

for right-shift 00s placed in high order bits. for left-shift 00s placed low order bits

left shift multiplying by 2^n

right shift divide by 2^n

does not exceed 255 though, shifting left will bring you back around

also lose fractional parts

bitwise logical operators- perform logic across bytes:

AND

```
 1110
&1010
```

```
-----
 1010
```

inclusinve OR

```
 1110
|1010
```

```
-----
 1110
```

exclusive OR

```
 1110
^1010
```

```
-----
 0100
```

complement operator- unary operator that reverses every bit in operand

bit fields in structures- structure field that contains specified number of bits. can have field with 1, 2 or 3 bits (lose advantage over 3 bits as might as well use int)

can store 8 yes or no values in single char

must be listed in structure first

specify size of field in bits following member name with colon and number of bits

```
struct emp_data
{
```

```
unsigned dental  :1;
unsigned college :2;
char fname[20];
char lname[20];
};
```