Developing Technical Software

Week 1:
Revisit Technical Programming –
Arrays and Pointers, Dynamic memory allocation

SWINBURNE UNIVERSITY OF TECHNOLOGY

V1.0_October 2018

Teaching Structure

Activity	Total hours	Hours per week	Teaching period
Lectures	28 hours	2 hour	Week 1 to 13
Tutorials	28 hours	2 hours	Week 1 to 13
Laboratory work	28 hours	2 hours	Week 1 to 13

Assessment

Tasks and Details	Individual or Group	Weighting	Due Date
Lab Assignment	Individual	30%	Week 1 to 13
Assignment 1	Individual	25%	Week 6
Assignment 2	Individual	25%	Week 10
Lab Test	Individual	20%	Week 12

Textbook

- Etter, D. M., *Engineering problem solving with C*, 4th edition, Pearson Prentice Hall, Upper Saddle River, New Jersey, 2013.

Recommended reference

- Deitel, H. M. & Deitel, P. J, *How to program*, 7th edition, Pearson Prentice Hall, Upper Saddle River, New Jersey, 2013.

Things to Remember

- Read syllabus/unit outline and understand learning outcomes/assessments
- Submit weekly lab tasks (Due date for every week is one week after the tutorial which means first week lab tasks should be submitted by the end of second week).
- Format of Weekly Lab tasks, Assignments should include Question, Code and Screenshot of the output in **one single word file.**
- Assignment weeks (Due dates are Week 6 for Assignment 1& Week 10 for Assignment 2)
- Demonstration of work (both assignment and labs) are mandatory – Zero marks will be awarded for poor demonstration or absent
- Lab Test – Week 12
- Never share your code with your friends – Plagiarism will not be tolerated.
- Should score minimum of 50% to pass the unit

Revision - Programming

- Programming is a series of instructions – a sequence of separate small commands one after the other
- **Sequence is vital in programming**
- For Example, one of your friend asks you directions over phone from Glenferrie Station to library. You would say
    1. Take left from Glenferrie Station
    2. Walk 100 mts
    3. Take left again on Burwood road
    4. Walk 300 mts
    5. Take left
    6. Walk 30mts and you will find library on the right
- Wrong sequence produce different results
- Sequential instructions are very important in programming

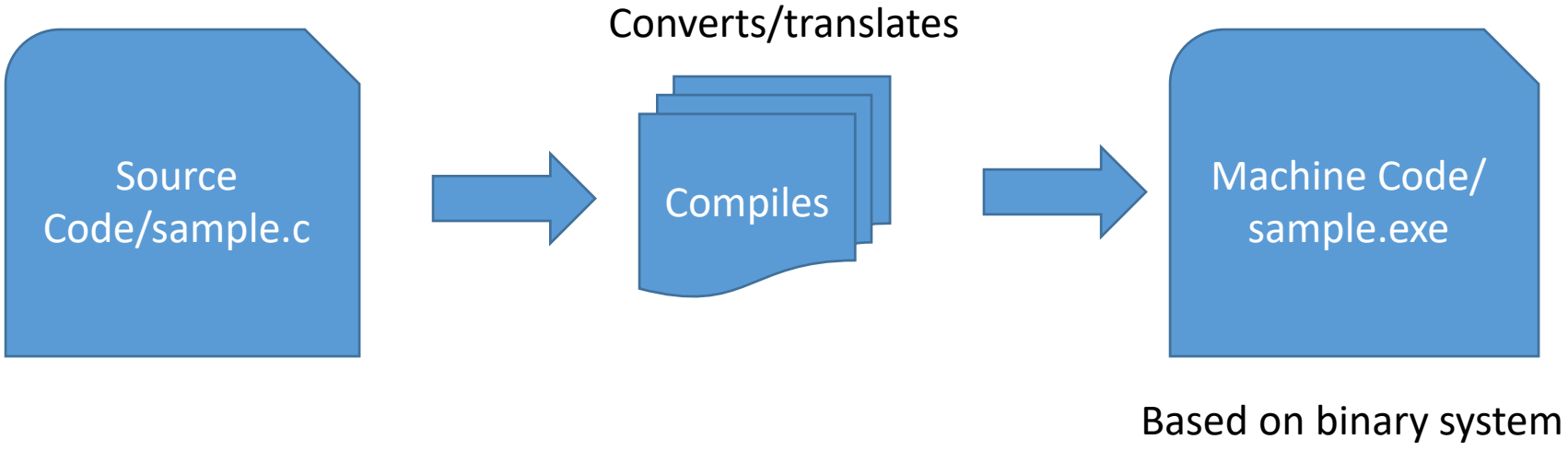
Revision – Central Processing Unit

- The Central Processing Unit (CPU) aka chip is the brain of the computer and only understands machine language based on binary system (0 and 1)
- Its is an electrical device and works on the logic of 0-1 within an electrical circuit
- For example, If current is flowing we can say its one, if it is not we can say its zero
- The machine code is unreadable and are tiny numerical instructions based on 0 and 1

Image Source:
https://www.intel.com.au/content/www/au/en/products/processors/core/i7-processors.html

Revision – Compilation

- As computer only understands machine language based on binary code we need to convert our code in such a way computer understands
- This conversion/translation process is called compilation



Revision – Memory

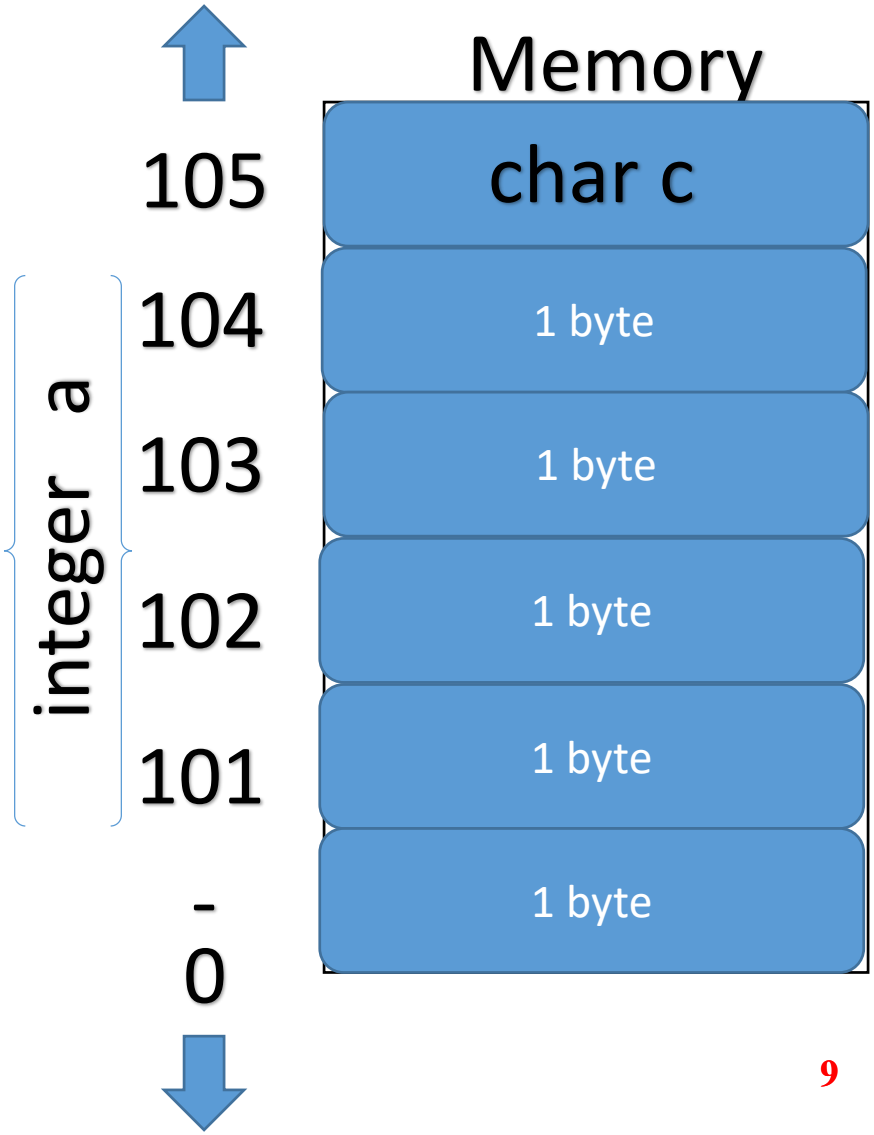
- There are two types of memory: hard drive and RAM
- All the files in your computer are typically stored in a hard disk drive which is called secondary storage
- Every program is executed by fetching the sequence of instructions from **RAM (Random Access Memory)** which is a primary storage or **main memory**
- RAM is a **volatile memory** which means everything will be **wiped off automatically** when the machine is turned off
- When program finishes execution, the memory in the RAM is cleared

Internal structure of Memory

Computer maintains an internal structure look-up table of where a particular variable is stored and which datatype is used

- Data are stored in blocks of memory
- Each block of memory is 1 byte
- Each byte has an address

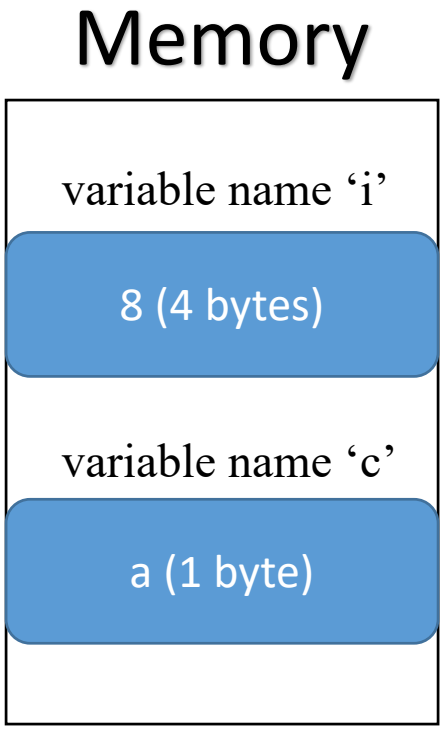
integer – 4 bytes
char – 1 byte
float – 4 bytes



Revision – Variables & Data Types

- Variables are used to store and represent the data
- The data which we are storing can be of different types such as character, numeric, decimal etc – data type

```
int i =8;
char c = 'a';
```



Arrays

- An array is a collection or systematic arrangement of variables of the **same** data type.
- If we have an array to store 10 integers, we may declare the array as follows:

```
int s[10];
```

10 integers are stored as `s[0], s[1], … , s[9].`

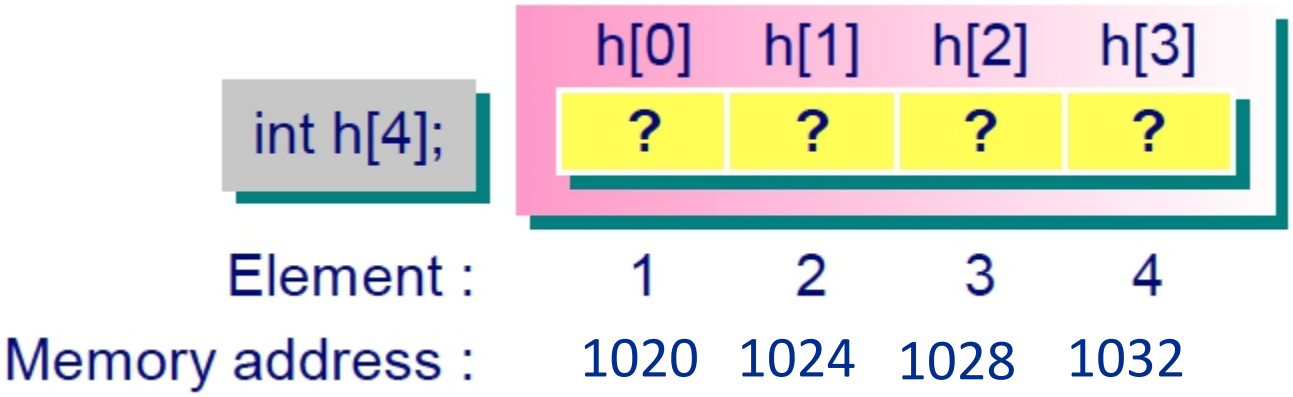
- The elements are stored in **contiguous/sequentially** in memory.

Declaration of arrays without initialization

```
float temp[365]; /* array of 365 floats */
char letter[26]; /* array of 26 chars */
int  number[5];  /* array of 5 integers */
```

- When an array is declared, n **consecutive** memory locations are allocated by the compiler for the whole array (e.g., 4 bytes for an integer)

```
int h
```



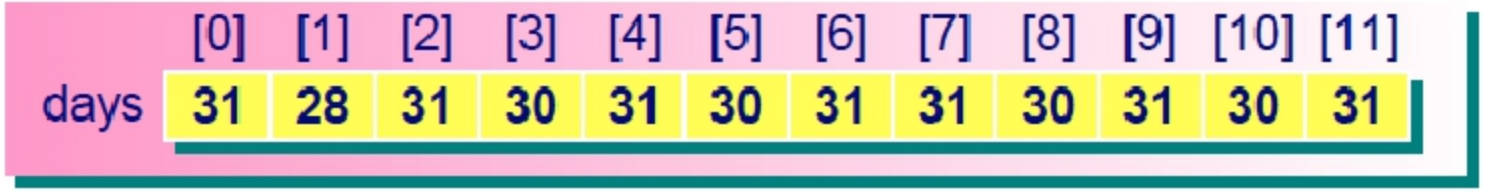
- The size of array must be integer constant.
- The element indices range **from 0 to n-1** where n is the declared size of the array

```
char letter[26];
letter[26] = 'z'; /* index out of range */
```

Initialization of arrays

- Initialize array variables at declaration

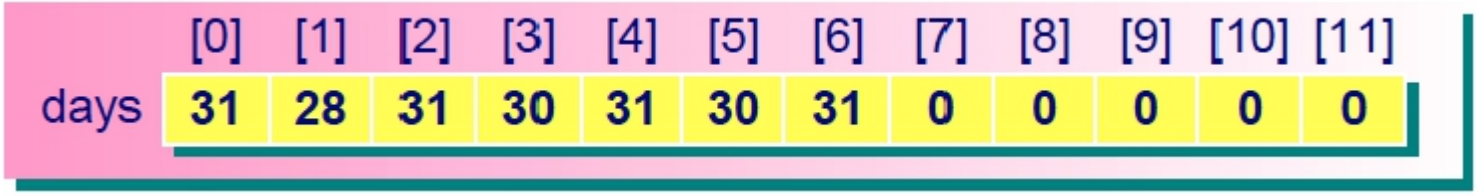
```
#define MTHS      12      /* define a constant */
int days[MTHS]={31,28,31,30,31,30,31,31,30,31,30,31};
```



- Partial array initialization

```
#define MTHS 12

int days[MTHS]={31,28,31,30,31,30,31};
```



Initialization of arrays (cont.)

- If an array is defined without a size, but with an initialization sequence, the size is defined to be equal to the number of values in the sequence:

```
char s[]={'A', 'b', '0'};
double t[]={0.0, 0.1, 0.2, 0.3};
```

- The size of an array must be specified in the declaration by using either a constant within brackets or by an initialization sequence within braces.

- Arrays can also be initialized with program statements:

```
int k;
double g[21];

for (k=0; k<=20; k++)
      g[k] =k*0.5;
```

Example 1.1: Find the average of ten numbers

```
#include <stdio.h>
int main( )
{
  int sample[10], i, total;
  for (i = 0; i < 10; i++) {
    printf (“enter number %d:”, i);
    scanf (“%d”, &sample[i]);
  }
  total = 0;
  /* add up the numbers */
  for (i = 0; i < 10; i++)
    total = total + sample[i];
  printf (“The average is %d\n”, total/10);
  return 0;
}
```

Example 1.2: Initialize the elements of an array to the even integers from 2 to 20

```
#include <stdio.h>
#define SIZE 10
int main()
{
  int s[SIZE], j;
  for (j=0; j<=SIZE-1; j++)
       s[j]=2+2*j;
       printf("%s %13s\n", "Element", "Value");
       for (j=0; j<=SIZE-1; j++)
  printf("%7d %13d\n", j, s[j]);
  return 0;
}
```

Element	Value
0	2
1	4
:	:
9	20

Multi-dimensional arrays (matrices)

- C also allows multi-dimensional arrays. A two-dimensional integer array `twod` with 10 rows and 20 columns can be declare as:

```
int twod[10][20];
```

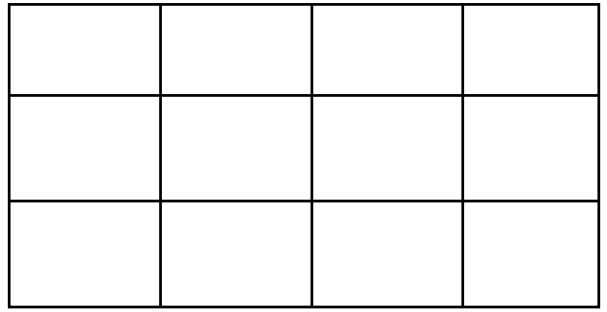
- Initialization of two-dimensional arrays can be done in many ways:

```
int twod1[3][2]={{2, 3}, {1, -5}, {0, 8}};
int twod2[4][3]={{1,2,1},{3,6,4},{2,7,6},{8,1,3}};
int twod2[][3]={{1,2,1},{3,6,4},{2,7,6},{8,1,3}};
```

Multi-dimensional arrays (cont.)

Example 1.3: Load the numbers in a two-dimensional array

```
#include <stdio.h>
int main()
{
  int i, j, num[3][4];
  for (i = 0; i < 3; ++i)
    for (j = 0; j < 4; ++j){
      num[i][j] = (i*4) + j + 1;
      printf("%d",(i*4)+j+1);
    }
  return 0;
}
```



Variable-length arrays

- C standard allows us to handle arrays of unknown size using variable length arrays (VLAs). These are not arrays whose size can change, but is defined in terms of an expression evaluated at execution time.
- * This is not supported by MS Visual C++ & Quincy

```
int main()
{
  int size;
  printf("Enter size of an array\n");
  scanf("%d", &size);
  int array[size]; /*declare 1-D VLA*/
}
```

Pointers

- Pointers are addresses!!!

In a computer, variables are stored in memory locations, and the memory location is assigned an address. We may have variables that store the addresses of memory locations of some data objects. These variables are called ***pointers***.

For example, in the following input statement

```
scanf("%d %d", &num1, &num2);
```

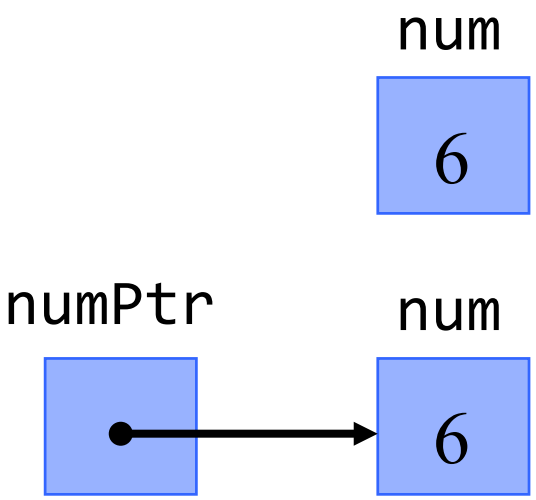
two variables read from the keyboard are assigned two addresses to store their values by using the address operator `&`, respectively. If you let

```
ptr = &num1;
```

the variable `ptr` is a pointer, storing the address of `num1`.

Referencing a variable

- A variable directly contains a specific value
- A pointer contains an address of a variable that contains a specific value.
- Referencing a value through a pointer is called *indirection*.



`num` directly references a variable that contains the value 6

Pointer `numPtr` indirectly references a variable that contains the value 6

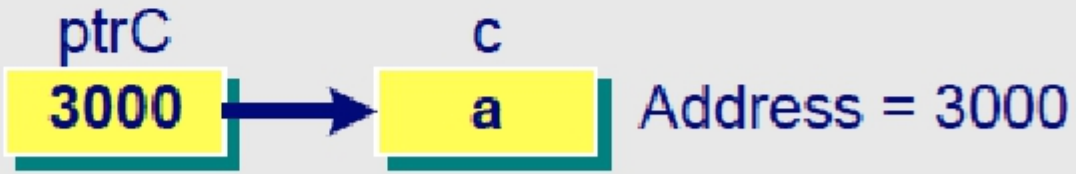
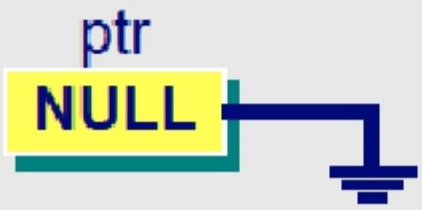
Pointer variable definition and initialization

- A pointer variable is declared by, for example

```
int  *ptrI;   /* The pointer ptrI is an address of an int */
char  *ptrC;  /* The pointer ptrC is an address of a char */
float *ptrF;  /* The pointer ptrF holds the address of a float */
```

- The above pointer variable declaration consists of two parts: (i) an asterisk (*) and (ii) the pointer variable name.

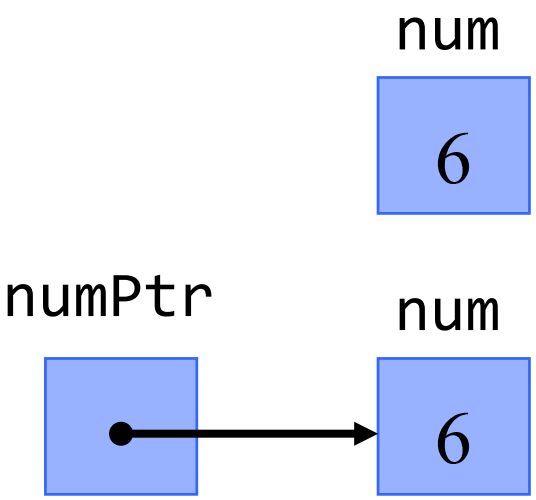
- The value of a pointer variable is an address, and should be initialized to `NULL` or an address.

Statement	Operation
int *ptrI ;	
ptrI = &a;	
ptrF = &b;	
ptrC = &c;	
int *ptr = NULL;	

Pointer operators

- **Address operator** (&) returns the address of its operand.
- **Indirection operator**, or dereferencing operator, (*) returns the content of the memory location pointed by a pointer variable.

```
int *numPtr, num;
num = 6;
numPtr = &num;
printf("%d", *numPtr);
```

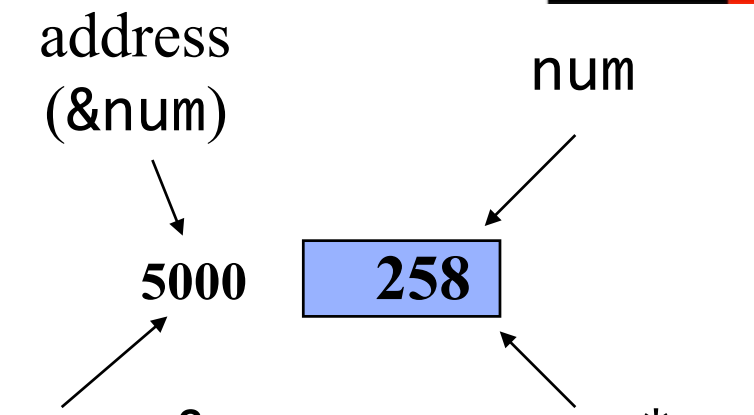


Example 1.4

```
#include <stdio.h>

int main ( )
 {
   int num=258;
   int *pa;
   pa=&num;
   printf("num=%d\n", num);

   printf("pa=%4d\n", pa);
   printf("*pa=%d\n",*pa);
   return 0;
 }
```



num = 258

pa = 5000

*pa = 258

Pointer assignments

```
int a, b, *ptr;
```

- When a pointer is defined, the type of variable to which it will point must also be defined.
- We need to specify an address to be stored in `ptr`, which can be made by a pointer assignment `ptr=&a`
- A pointer can store an address of another pointer

```
int **ptrptr;
int a, b, *ptr;

    ptr=&a
    ptrptr = &ptr
```

Void pointer

```
int a;
float b;
void *ptr;
```

- A void pointer `ptr` is a specific type of pointer that can point to objects of any data type: `ptr=&a, ptr=&b`
- However, since the void pointer does not know what type of object it is pointing to, it cannot be dereferenced. Thus, `*ptr` is not valid operator.
- To access the data pointed to by a void pointer, we typecast it with the correct type of the data held inside the void pointer location, e.g. `*((int*)ptr), *((float*)ptr)`

Example 1.5:

```
#include <stdio.h>
 int main()
 {
   int   x;
   int *p1, *p2;
   x   =   101;
   p1   =   &x;
   p2 = p1;
   printf("at location %d\n", p2);
   printf("is the value %d\n", *p2);
   return 0;
 }
```

at location 6000
is the value 101

Pointer Arithmetic

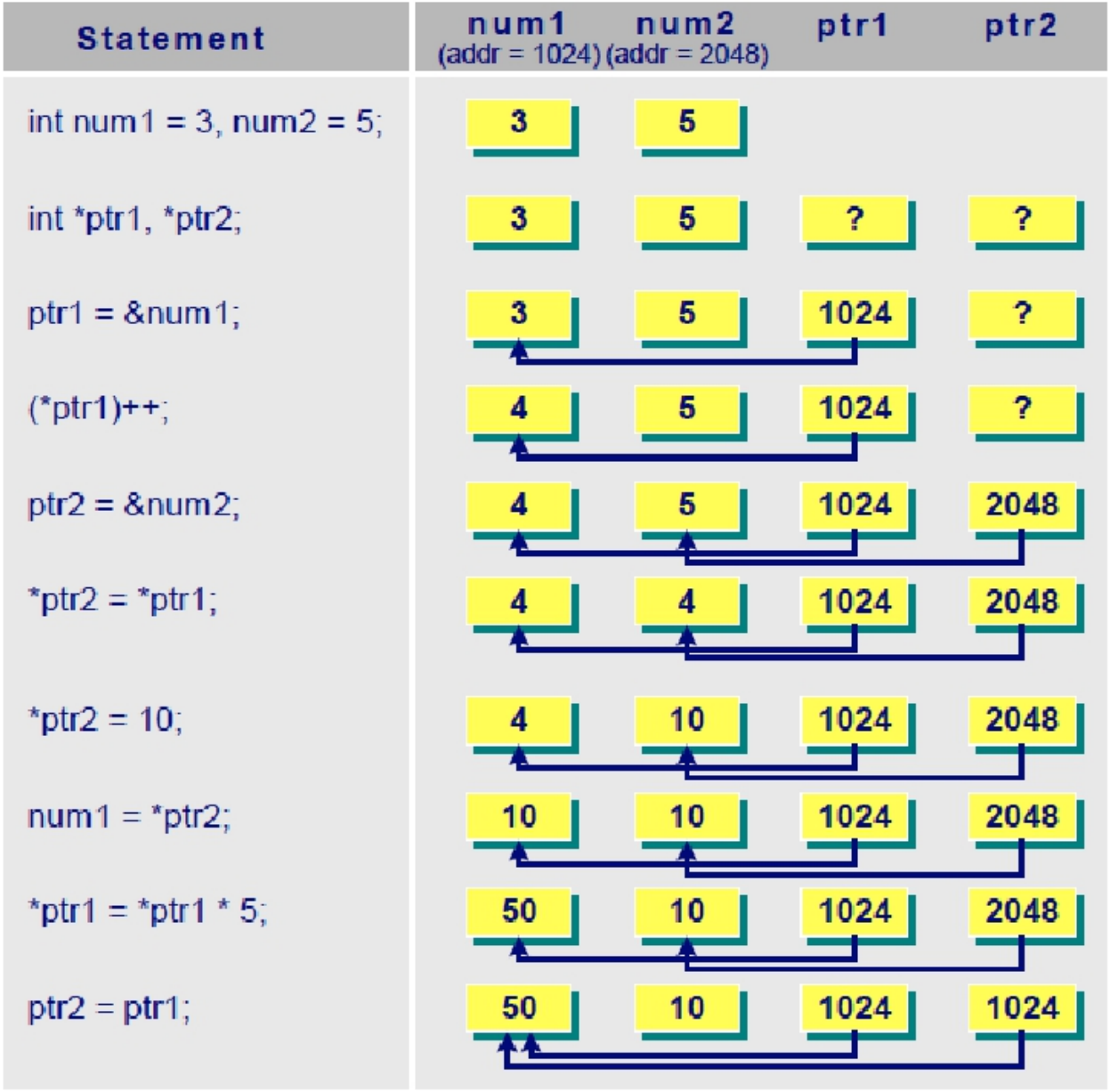
The operations that can be performed with pointers are limited to the following

- A pointer can be assigned to another pointer of the same type
- An integer value can be added to or subtracted from a pointer
- A pointer can be assigned to or compared with the integer zero

or to the symbolic constant NULL, which is defined in `<stdio.h>`

```
int **ptrptr;
int a, b, *ptr1, *ptr2;

    ptr1=&a;
    ptr2=NULL;
    ptr1++;
    ptrptr = &ptr1;
    ptrptr +=2;
```



How are pointers related to arrays?

Consider an inter is 4 bytes, and assume the declaration:

```
int a[3];
```

The array name, **a**, is a pointer which points to the address of the first element
`a[0]` of the array.

Example 1.6: Pointers and 1-D arrays

```
#include <stdio.h>
int main()
{
int *px, *py;
int a[10] = {1,2,3,4,5,6,7,8,9,10};
px = &a[0];
py = a;
printf("px=%d\n py=%d\n a=%d\n a[0]=%d", px,py,a,a[0]);
px = px+1;
printf("px+1=%d\n &a[1]=%d\n", px, &a[1]);
return 0;
}
```

px = 4202504
py = 4202504
a = 4202504
a[0] = 1

px+1=4202508
&a[1]=4202508

Dynamic memory allocation

- If we don't know an array's size at compilation time, we would have to use dynamic-memory-allocation related functions, e.g. **`malloc`** and **`free`**, and **`sizeof`** operator
- For creating and maintaining dynamic data structures
- The ability for a program to *obtain more memory space at execution time*, and to *release space no longer needed*
- *Allocate block of memory: malloc, calloc, realloc*
- *Deallocate block of memory: free*

malloc()

- Frequently used library function for dynamic memory allocation
- It takes as an argument the number of bytes to be allocated and returns a pointer of type void * (pointer to void) to the allocated memory.

void *malloc(size_t number_of_bytes)

- This function as argument asks for the size of the memory block
- The data type size t is a data type that stores only positive integers
- The function malloc returns a void pointer that gives the address of the first byte of the newly created block
- Example:

```
char *cp;
cp = malloc(100);
```

attempts to get 100 bytes and assigns the start address to cp

sizeof()

- `malloc()` is usually used with the `sizeof()` operator.

```
int *ip;
ip = (int *)
malloc(100*sizeof(int));
```

- To find the size of any data type, variable or structure, e.g.

```
int i;
struct COORD {float x,y,z};
typedef struct COORD PT;
```

- Therefore `sizeof(int)`, `sizeof(i)`, `sizeof(struct COORD)` and `sizeof(PT)` are all acceptable
- We can use the link between pointers and arrays to treat the reserved memory like an array. *For example*:

```
ip[0] = 100;
```

- or

```
for(i=0;i<100;++i) scanf("%d",ip++);
```

free()

- When you have finished using a portion of memory you should always free it.
- This allows the memory *freed* to be available again, possibly for further `malloc()` calls
- The function `free()` takes a pointer as an argument and frees the memory to which the pointer refers.

```
free(ip);
```

Resources

1. https://www.youtube.com/watch?v=5tPLyHCZdU0
2. https://www.youtube.com/watch?v=h-HBipu_1P0&t=376s

Thank you

End of week 1