module5

gar

Outline

Pointers

- Pointer is a variable that contains the address of a variable
- The memory addresses are numbered consecutively
- A `char` variable takes one byte of memory, and can be located anywhere in memory
- A `short int` is stored in a pair of adjacent bytes of memory
- An `int` is stored in four adjacent bytes of memory
- In 32-bit addressable systems, `long` also takes four bytes
- In 64-bit addressable systems, `long` takes eight bytes
- This piece of information is critical when dealing with pointers

Syntax to use Pointers

- ```
  char ch = 'a';
  char *pc = &ch;
  printf("%c, %u, %c\n", ch, pc, *pc)
  ```
- The above set of instructions print 'a' (the value in `ch`), the address of `ch`, and again 'a'
- Unary & operator gives the address of a variable
  - but binary & performs `bitwise-and` of two variables
- Unary * operator is called the dereferencing operator, which when applied to a pointer, it accesses the object the pointer points to
  ```

Examples on Pointers

- ```c
  char c = 'C';
  int i = 256;
  char *pc = &c;
  char *pi = &i;
  printf("%d\n", *pi);
  ```
- In the example, `char *pc` means that `pc` points to a character
- But `pi` also has been declared as a pointer to a character
- The compilation is possible, but a warning like "incompatible pointer type" will be issued
- Whatever variable a pointer points to, the size of all the pointers will be the same
  - because they contain address, which will be 32 bits or 64 bits depending on the compiler and the OS
- The incompatible pointer will cause problems when dereferencing
- In the example, 0 is printed instead of 256
  - because $256 = 2^8$, which means 100000000 in binary
  - and only the last 8 bits are accessed since we have declared it as a pointer to a `char`
  ```

Pointers and Function Arguments

- Pointers can be used to access elements in another function
- Consider a function `inc5` supposed to increment a variable by 5

```
void inc5(int i)
{
  i = i+5;
}
```

- If we call `inc5(a)`, variable a will not be incremented since a copy of a is passed (pass by value)
- Now, modify the function

```
void inc5(int *i)
{
  *i = *i+5;
}
```

- If we call `inc5(&a)`, variable a will be incremented since address of a is passed (pass by reference)

Pointers and Arrays

- An array a defined as

```
int a[5] = {10, 20, 30, 40, 50};
```

  will contain the address of the first element
- a[1] will access the next element
  - which is same as *(a+1)
- We may define a pointer to array

```
int *pa = &a[0]; /* points to first element of a */
int *pa2 = a;     /* also means the same */
```

- The only difference is that the pointer is a variable, whereas the array name is not
  - So, pa++; will point to the next element, but a++; is an error since a cannot change

Address Arithmetic

- If pa is a pointer to an array, then `pa++;` increments the value of pa such that it points to the next element in the array
- A small program written and run in a computer will verify it

```
int main ()
{
  int a[5] = {1,2,3,4,5};
  char c[] = "tring";
  int *pa = a;
  char *pc = c;
  printf("%u %u\n", pa,pa+1); /* 3899203328 3899203332
  printf("%u %u\n", pc,pc+1); /* 3899203360 3899203361
  return 0;
}
```

Character Pointers and Functions

- We may also have a pointer to a string

```
char a[] = "a string";
char *ps = "yet another string";
```

- a is an array name, which contains the starting address of the string
- The string will be stored somewhere in memory, and its starting address will be assigned to ps

Character Pointers and Functions

- Write a function `strlen` to compute the length of a string using pointers

```
int strlen(char *s)
{
    int n;
    for (n = 0; *s != '\0'; s++)
        n++;
    return n;
}
```

- We may then call the function in multiple ways:

```
strlen("hello, strlen"); /* string constant */
strlen(a);               /* char a[]; */
strlen(ps);              /* char *ps */
```

Character Pointers and Functions

- Write a function `strcpy` to copy a source string to destination string

```
void strcpy(char *s, char *t)
{
    while ((*s = *t) != '\0') {
        s++;
        t++;
    }
}
```

- which can be equivalently shortened to

```
void strcpy(char *s, char *t)
{
    while ((*s++ = *t++) != '\0') ;
}
```

Pointer Arrays

- Since pointers are variables, they can also be stored in arrays
- One of the useful applications of such an array is to sort names

```c
char *ps[] = {"b1", "a12", "c3"};
```

- The three strings are in different memory locations, and the pointer array holds addresses in the order of the strings
- Now, to sort names, instead of swapping the complete string character by character, we only swap the first two addresses in the pointer array

Dynamic Memory Allocation

- Sometimes, the size of the input data will be unknown in advance
- Allocating the maximum possible size may waste a lot of space
  - E.g. If we are going to get an array of up to 1000 integers, then declaration like `int num[1000];` can be made,
  - But, it wastes space if we get lesser number of elements
- So, dynamic memory allocation is used, which will allocate space for the variables when it's required

Dynamic Memory Allocation: `malloc`

- `malloc` is one function which can allocate space as needed
- Its prototype is described in `stdlib.h`
- Its usage is:

```
ptr = (type) malloc(size);
```

  - `malloc` returns a pointer to the memory allocated
  - It's of type `void *`, called a *generic pointer* and must be explicitly typecast to the appropriate data type

Dynamic Memory Allocation: `malloc` example

```
struct emp {
    char name[20];
    int empnum;
    double salary;
};
struct emp *worker;
worker = (struct emp *) malloc(sizeof(struct emp));

worker -> empnum = 1;
```

- `sizeof` returns the required number of bytes for the structure
- `malloc` reserves the space and returns the address of the space, which is converted to the structure data type

Array of Pointers

- If more than one employee is information is to be stored, an array of pointers can be declared

```
struct emp {
     char name[20];
     int empnum;
     double salary;
};
struct emp *worker[20];
worker[3] = (struct emp *) malloc(sizeof(struct emp));

worker[3] -> empnum = 4;
```

- Whenever a new employee is hired, the index value is incremented, which then points to the new employee

Freeing Memory

- After using the allocated memory, we need to `free` it so that it can be re-used
- Its general form is

```
free(ptr);
```

- `ptr` must be pointing to some memory address

Preprocessor Directives

- Just before compiling a program, it involves a preprocessing stage
- Preprocessor modifies the source code before it is handed over to the compiler
- Such modifications are indicated by preprocessor directives, which are indicated by # symbol
- It provides the ability for the inclusion of header files, macro expansions, conditional compilation, etc.

#define directive

- #define directive is used to substitute some text in the source code
- It's also called a macro
- The syntax is

  #define identifier <substitute text>

- Example

  #define PI 3.14159265359

  - Replaces every occurance of `PI` with the defined value

(Extra) Checking the Preprocessor Output

- gcc provides with an option `-E`, which enables to see the output of the preprocessing stage
- E.g.

```c
/* store program as pi.c */
#define PI 3.14159265359

int main()
{
  printf("%f\n", PI/2);
  return 0;
}
```

  - `gcc -E pi.c`
- Check the output and observe the changes

Macros with Arguments

- Macros can also receive parameters
- E.g.

```c
#define DOUBLE(a) (a)*2

int main()
{
    printf("%d", DOUBLE(5+3));
    return 0;
}
```

  - `DOUBLE(5+3)` is substituted with (5+3)*2 after preprocessing

Undefining a Macro

- A macro defined with #define can be undefined using #undef directive
- That macro can then not be used after undefining
- Useful when we are trying to redifine a macro to a new value
- E.g. to assign an approximate value of pi to a macro `M_PI` defined in `math.h`

```
#include <math.h>
#include <stdio.h>

#undef M_PI
#define M_PI (22/7.0)
```

#include directive

- The #include directive loads the specified file in the program
- The included file is also compiled with the program
- Two ways of including

```
#include <filename> /* 1 */

#include "filename" /* 2 */
```

- The header files stored in standard directories will be included if < > is used
- The header files stored in current and standard directories will be included if " " is used

Conditional Compilation

- The statements are compiled only if some condition is true
- Syntax

```
#ifdef <identifier>
{
statements;
}
#else
{
statements;
}
#endif
```

Conditional Compilation

- E.g.

```
#include <stdio.h>
#include <math.h>
#define E =
int main()
{
    int a;
    #ifdef E
    {
        a E 1;
    }
    #else
    {
        a = 2;
    }
    #endif
    printf("%d", a);
    return 0;
}
```

Data Structures

- Data Structure is a method of storing data in a computer so that it may be used efficiently
- Data Structure
  - Primitive
    - `int`, `char`, `float` etc.
  - Non-Primitive
    - Linear (Arrays, stacks, queues, linked-lists)
    - Non-linear (Trees, graphs)
- The basic data type provided by the programming language is called the primitive data type
- The data type derived from basic types is called non-primitive data types

Stack

- Stack is a data structure where the elements are inserted to one end and deleted from the same end
- The position where the insertion and deletion happens is called the top of the stack
- Also called Last-In-First-Out (LIFO) data structure
- Three main stack operations
  - Push
  - Pop
  - Display

Queue

- Stack is a data structure where the elements are inserted to one end and deleted from the other end
- The end where the elements are inserted is called the rear end
- The end where the elements are deleted is called the front end
- Also called First-In-First-Out (FIFO) data structure
- Operations on queues
  - Insert (Enqueue)
  - Delete (Dequeue)
  - Display

Linked List

- A data strucutre which is a collection of zero or more nodes where each node has some information
- Node consists two fields
  - `info`: which holds some information
  - `link`: contains the address of the next node
- Types of linked list
  - Singly linked lists
    - Last node's `link` field will be `NULL`
  - Doubly linked lists
    - Contains two `link` fields, for right and left nodes
  - Circular singly linked lists
    - Last node's `link` field will contain the address of the first node
  - Circular doubly linked lists

Singly Linked Lists

- Operations on singly linked lists
  - Inserting a node
  - Deleting a node
  - Search in a list
  - Display the contents

Trees

- Non-empty set of items where one element is called a root, and the remaining items are divided into $n \geq 0$ disjoint subsets, each of which can be a tree
- Every item is called a node
- Every node can have zero or more branches (subtrees)