CS 2412 Data Structures

Chapter 1 Introduction

1.1 Basic concepts

The basic idea of computer is to treat data: data input, data storage, data treatment, and data output.

Data Structure

1. A combination of elements in which each is either a data type (a set of values and operations on values) or another data structure
2. A set of associations or relationships involving the combined elements

Data structures and algorithms–the basic elements from which large and complex software are built.

To understand a data structure requires following things:

- learn how the information is arranged in the memory of the computer.
- familiar with the algorithms for manipulating the information contained in the data structure.
- understand the performance characteristics of the data structure so that when called upon to select a suitable data structure for a particular application.

Abstraction

It is said that "computer science is the science of abstraction." But what exactly is abstraction? Abstraction is "the idea of a quality thought of apart from any particular object or real thing having that quality".

Abstraction is used to suppress irrelevant details while at the same time emphasizing relevant ones. The benefit of abstraction is that it makes it easier for the programmer to think about the problem to be solved.

ADT

Abstract data types:

1. a set of values (declaration of data)
2. a set of operations (declaration of operations)
3. encapsulation of data and operations

For example, when we declare a variable, say x, of type `int`, we know that x can represent an integer in the range $[0, 2^{32} - 1]$ (assuming 32-bit integers) and that we can perform operations on x such as addition, subtraction, multiplication, and division. The type `int` is an abstract data type in the sense that we can think about the qualities of an `int` apart from any real thing having that quality. In other words, we don't need to know how `int`s are represented nor how the operations are implemented to be able to use them or reason about them.

Structured programming

Top-down refine design programs and projects.

Top-down specifications of variables.

Pascal and C are structured programming languages. While Java and C++ are object-oriented programming languages.

1.2 C reviews

Variables

A C variable has attributes such as name, type, value, address, size, lifetime and scope.

Passing

There are two parameter passing mechanisms in C:

1. pass-by-value
2. pass-by-reference.

Both of these methods are used extensively. It is essential that you understand the behavioral difference between the two methods as well as the performance implications of using each of them.

Pointer

Mastering the use of pointers is essential when programming in C. The key to understanding pointers is to recognize that a pointer variable has exactly the same set of attributes as any other C variable. It is crucial that you keep straight the distinctions between the value of a pointer, the address of a pointer and the object to which a pointer points.

Example:

```
#include <stdio.h>
int main() {
int num=22;
printf("num is %d, the address of num is %p \n",num,&num);
printf("The address of the integer next to num is %p \n",
&num+1);
}
```

output: `num is 22, the address of num is ffbff284`

`The address of the integer next to num is ffbff288`

Note: An integer uses 4 bytes. An address is a hexadecimal. In C two important attributes of a variable are: the value stored in it and the address of it.

Example:

```
#include<stdio.h>
void func(int *) //prototype
int main() {
int x=5,*pi;
void func(int *);
printf("x = %d \n",x);
pi=&x;
func(pi);
printf("new x = %d \n",x);
}
```

The source code of `func` is:

```
void func(int *py){
*py=(*py)*(*py);
}
```

Note:

- The output is: `x = 5` and `new x = 25`.
- In declaration `*py` means `py` is a pointer, while in `func` body `*py` means the value at the location referenced by the pointer `py`.
- A declaration of a pointer specifies the data type to which the pointer points to. If `pi` is a pointer of integer, then `pi+1` points to an integer located next to the integer pointed by `pi`. So `*pi+1` and `*(pi+1)` are different.
- Dereference a pointer before it was initialized is dangerous.
- The example shows how to use a pointer to send the address to a function.

Pointer to `void`

A generic pointer which can be assigned without a cast. It can be used to represent any data type.

```
#include<stdio.h>
int main ()
{
 void* p;
 int i = 7;
 float f = 23.5;
 p = &i;
 printf(``i contains: %d\n'', *((int*)p) );
 p = &f;
 printf(``f contains: %d\n'', *((float*)p) );
 return 0;
}
```

Note

A pointer to void cannot be dereferenced. In other words, we cannot use `*p` without cast. In the previous program, the code in `printf` the pointer `p` must be cast to a type.

The function `malloc` returns a pointer to `void`. So when the function is called, it is recommended that the returned pointer be cast to the appropriate type.

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

Pointer to function

The name of a function is a pointer constant to its first byte of memory. So we can define a pointer to a function.

Suppose we have three functions:

```
void fun (void)
{...
}
int pun (int, int)
{...
}
double sun (float)
{...
}
```

We can define pointers as follows:

```
void (*f1) (void);
int  (*f2) (int, int);
double (*f3) (float);
... ...
f1 = fun;
f2 = pun;
f3 = sun;
... ...
```

Example

Suppose we want to write a C program to compare two numbers. First we give an simple algorithm:

```
1. input two numbers.
2. compare the two numbers.
3. print out the larger number.
```

Then we need a more details about step 2: how to compare two numbers. We may consider two things:

- To compare two numbers x_1 and x_2, it is better to compute if $x_1 - x_2 > 0$.
- The type of the two numbers is not fixed. So we can compare two integers, or compare two real numbers, etc.

```
void* larger (void* dataPtr1, void* dataPtr2,
                    int (ptrToCmpFun)(void*, void*))
{
if ((*ptrToCmpFun) (dataPtr1, dataPtr2) > 0)
     return dataPtr1;
else
    return dataPtr2;
}
```

Compare two integers

```
#include <stdio.h>
#include <stdlib.h>
#include ‘‘P1-0.5.h’’ //local header file
int compare (void* ptr1, void* ptr2);

int main (void)
{
int i = 7;
int j = 8;
int lrg;
lrg = (*(int*) larger (&i, &j, compare));
printf(‘‘Larger value is: %d\n’’,lrg);
return 0;
}
```

```
int compare (void* ptr1, void* ptr2)
{
if (*(int*)ptr1 >= *(int*)ptr2)
  return 1;
else
  return -1;
  }
```

The `larger` function can also be used to compare two floating-point values.

Array and pointer

Declare arrays:

```
int a[20],b[],c[3][2];
```

What are the meanings of the following variables?

```
a[0], a, c[0][1], c[0], c
```

Note: A name of an array is a pointer.

Example:

```
#include<stdio.h>
int main() {
int a[10]={1,2,3,4,5,6,7,8,9,10};
char name[9]={'L','a','k','e','h','e','a','d','\0'};
char course[]="Data structure";
printf("the 3rd number of a is %d. \n",*(a+2));
printf("%c \n", name[0]);
printf("%s \n", course);
printf("%s \n", name);
}
```

What are the output?

Example:

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

What are the output?

Passing arrays

Example:

```
#include<stdio.h>
int findMax(int[],int);
int main(){
int nums[5]={2,18,1,27,16};
printf("The maximum value is %d. \n",findMax(nums,5));
return 0;
}
int findMax(int *vals, int numEls){
int i, max=*vals;
for(i=1;i<numEls;i++)
 if(max < *(vals+i)) max=*(vals+i);
return (max);
} //also can use vals[] instead of *vals
```

Structures

Two basic ways to declare a structure.

```
struct
 { int day;
   int month;
   int year;
  } birth,*graduate;


struct date
 { int day;
   int month;
   int year;
  };
struct date birth,*graduate;
```

Using typedef

```
typedef struct
 { int day;
   int month;
   int year;
  } date;
date birth,*graduate;
```

To initialize a structure:

```
birth={10,3,1980};
```

or

```
birth.day=10;
......
graduate->day=1;
graduate->month=5;
graduate->year=2003;
```

Structure arrays

```
#include<stdio.h>
#define NUM 50
typedef struct
{ char name[20];
  long id;
  char major[10];
} sign;
```

```
int main(){
int i;
sign cs2412[NUM];
for(i=0;i<NUM;i++)
 { printf("Please input your name,id and major \n");
  scanf("%s,%d,%s",cs2412[i].name,&cs2412[i].id,
  cs2412[i].major);
  }
for(i=0;i<NUM;i++)
printf("name: %s,ID: %d, major: %s\n", cs2412[i].name,
cs2412[i].id,cs2412[i].major);
}
```

Structure of structures

```
struct data
{ sign info;
  float grade;
  char email[50];
} student;
```

Passing and returning structures

Example

```
#include<stdio.h>
#define NUM 30
typedef struct
 { char name[20];
   float assignment;
   float midterm;
   float final;
 } grade;
float average(grade *)
```

```
int main(){
grade cs2412[NUM];
float average(grade *),aver[NUM];
int i;
for(i=0;i<NUM;i++)
 { printf("Please input name, grades of assignment,
  midterm test and final exam:\n");
  scanf("%s %f %f %f", cs2412[i].name,
  &cs2412[i].assignment,&cs2412[i].midterm,
  &cs2412[i].final);
  aver[i]=average(&cs2412[i]);
  }
for(i=0;i<NUM;i++)
printf("%s 's final grade is %3.1f \n", cs2412[i].name,
aver[i]);
}
```

```
float average(grade *ps)
{
 return(ps->assignment*0.3 + ps->midterm*0.2+
   ps->final*0.5);
}
```

The average is calculated with the weights 30 % of assignments, 20 % of midterm test and 50 % of final test.

Header files

When we develop a big program, usually we need to write own headers to avoid redundancy. Following preprocessor directives are used in header files:

`#include, #define, #ifndef, #endif, #if`

Suppose we want to write a header file `myheader.h`. The file may contain following items.

```
#ifndef _MYHEADER_H
#define _MYHEADER_H
/*include files*/
#include<stdio.h>
#include ...
......
/* global variables, data types, etc*/
typedef struct
  { ...
    ...
  } Date;
#define PI 3.114159
 ......
```

```
/*prototype of functions*/
int func1(int *,float);
void func2(int[],int0;
......
#endif
```

If _MYHEADER_H is already defined, then all the statement in this file will not execute again. In this way, we can include the header file in each of the functions.

```
#include "myheader.h"
int main(){
 ......
 }
```

In source codes of functions:

```
#include "myheader.h"
 int func1(int *p, float a)
 {
  ......
 }
```

and

```
 #include "myheader.h"
 void func2(int arr[], int size)
 {
  ......
 }
```

Makefile

To compile files and make executable code, usually we need to create `makefile`. There are automate tools to create makefiles in many softwares such as `lcc`, visual studio etc. But very often we need to create or revise a `makefile`. Makefile is dependent on compiler and operating system. Main grammar rules are:

```
# comment line
Target file: dependency files (dependency line)
< Tab > actions (action line leaded by Tab key)
```

A simple makefile in UNIX system.

```
#makefile for ex2

ex2: ex2.o ex2_1.o
        gcc -o ex2 ex2.o ex2_1.o -lm
        rm *.o
ex2.o: ex2.c
        gcc -c ex2.c
ex2_1.o: ex2_1.c
        gcc -c ex2_1.c
```

Detailed rules for UNIX system can be found at

`/usr/share/lib/make/make.rules`

```
# Wedit Makefile for project row10-15
SRCDIR="c:\users\rwei\c program"
CFLAGS= -g2
CC=$(LCCROOT)\bin\lcc.exe
LINKER=$(LCCROOT)\bin\lcclnk.exe
OBJS=\
    row10-15.obj

LIBS=
EXE="row10-15.err.exe"

$(EXE): $(OBJS) Makefile
    $(LINKER)  -subsystem console -o $(SRCDIR)\ ...
```

```
# Build row10-15.c
ROW10-15_C=\

row10-15.obj: $(ROW10-15_C) $(SRCDIR)\"row10-15.c"
    $(CC) -c $(CFLAGS) $(SRCDIR)\"row10-15.c"

link:
    $(LINKER)  -subsystem console -o $(SRCDIR)\"...

clean:
    del $(OBJS) row10-15.err.exe
```

Some tips for programming

- Name the variables carefully and give detailed explanations.
- Give comments as much as you can.
- Each function should do only one task.
- Keep your input and output as separate functions.
- Keep in mind that your source codes are read by other people.

1.3. Algorithm efficiency

- The efficiency of an algorithm is based on the usage of time and space. We will basically focus on time efficiency.
- The efficiency can be considered in average cases and worst cases.
- Abstract methods are used to do the efficiency analysis. We consider the rate of growth and use asymptotic notation.

- A function or algorithm is linear, if there is no loops or recursions. The efficiency depends on the number of instructions.
- We will focus on algorithms with loops for their efficiency.
- We are concerned with how the running time of an algorithm increases with the size of the input in the limit, as the size of the input increases without bound.
- We will discuss the big O notation for this asymptotic aspect. There are other notations like Ω, Θ, etc which will be discussed in other courses.

Loops

We discuss the algorithm efficiency as a function of the number of elements to be processed (inputs). So we consider the efficiency as a function $f(n)$, where n is the number of inputs.

- Linear loops: `for (i = 0; i < 1000; i++) ...`. In this case, $f(n) = n$.
- Logarithm loops: `for (i = 1; i <= 1000; i*=2), ...`. Then $f(n) = \log n$.
- Nested loops: `for (i = 0; i < 10; i++) for (j = 0, j < 10; j++) ...`. Then $f(n) = n^2$.

Big-O notation

If there exist positive integers c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$, then we say that

$$f(n) = O(g(n))$$

The O notation gives an upper bound for the function.

Examples

- If $f(n) = 10n^2$, then $f(n) = O(n^2)$.
- If $f(n) = n^2 + n + 1$, then $f(n) = O(n^2)$.
- If $f(n) = n^2$, then $f(n) \neq O(n)$.
- ```
  for ( i = 0; i < 10; i++)
    for (j = 0; j < i; j++)
      application code ...
  ```
  Then the complexity $f(n) = O(n^2)$.

The seven standard measures of efficiency are
$O(\log n), O(n), O(n \log n), O(n^2), O(n^k), O(c^n)$, and $O(n!)$.
  ```