EECS Cheatsheet

Loops

1. The **`for` loop** is commonly used when you know the number of iterations beforehand. It has three parts: initialization, condition, and increment/decrement.

```
for (initialization; condition; increment/decrement) {
    // code to be executed
}
```

2. The **`while` loop** repeatedly executes a block of code as long as the specified condition is true. It's typically used when the number of iterations is not known beforehand.

```
while (condition) {
    // code to be executed
}
```

3. **Do-while Loop** : Similar to the `while` loop, but the condition is checked after the execution of the loop's body. This guarantees that the loop will execute at least once.

```
do {
    // code to be executed
} while (condition);
```

The key differences between these loops are:

- **Initialization and Increment/Decrement**: For loops have a built-in mechanism for initialization and updating loop control variable, making them ideal for iterating a specific number of times. While and do-while loops require these to be handled manually outside the loop.

- **Condition Checking**: In the while loop, the condition is checked before entering the loop body, so if the condition is initially false, the loop may not execute at all. In the do-while loop, the condition is checked after the loop body, so the loop body is always executed at least once.

- **Suitability**: For loops are typically used when you know the number of iterations beforehand. While loops are used when the number of iterations is not known and termination is based on a condition. Do-while loops are used when you want to execute the loop body at least once regardless of the condition.

Functions

7.1 A BRIEF OVERVIEW

A *function* is a self-contained program segment that carries out some specific, well-defined task. Every C program consists of one or more functions (see Sec. 1.5). One of these functions must be called `main`. Execution of the program will always begin by carrying out the instructions in `main`. Additional functions will be subordinate to `main`, and perhaps to one another.

If a program contains multiple functions, their definitions may appear in any order, though they must be independent of one another. That is, one function definition cannot be embedded within another.

A function will carry out its intended action whenever it is *accessed* (i.e., whenever the function is "called") from some other portion of the program. The same function can be accessed from several different

Generally, *a function will process information that is passed to it from the calling portion of the program, and return a single value.* Information is passed to the function via special identifiers called *arguments* (also called *parameters*), and returned via the `return` statement. Some functions, however, accept information but do not return anything (as, for example, the library function `printf`), whereas other functions (e.g., the library function `scanf`) return multiple values.

There are several advantages to this modular approach to program development. For example, many programs require that a particular group of instructions be accessed repeatedly, from several different places within the program. The repeated instructions can be placed within a single function, which can then be accessed whenever it is needed. Moreover, a different set of data can be transferred to the function each time it is accessed. Thus, *the use of a function avoids the need for redundant (repeated) programming of the same instructions.*

The use of functions also enables a programmer to build a *customized library* of frequently used routines or of routines containing system-dependent features. Each routine can be programmed as a separate function and stored within a special library file. If a program requires a particular routine, the corresponding library function can be accessed and attached to the program during the compilation process. Hence a single function can be utilized by many different programs. This avoids repetitive programming between programs. It also promotes *portability* since programs can be written that are independent of system-dependent features.

1. Functions are declared before they are used. The **declaration** typically includes the function's return type, name, and parameters (if any). The function definition contains the actual implementation of the function.

```
// Function declaration
int add(int a, int b);

// Function definition
int add(int a, int b) {
    return a + b;
}
```

2. **Function prototypes** provide a declaration of the function to the compiler without providing its implementation. They allow the compiler to know about the function's signature before it is used in the program.

```
// Function prototype
int add(int a, int b);
```

3. Functions can take zero or more parameters. **Parameters** allow you to pass data to the function, which it can then operate on.

```
int add(int a, int b) {
    return a + b;
}
```

4. Functions can return a value using the **`return` statement**. The return type of the function specifies the type of the value that the function returns.

```
int add(int a, int b) {
    return a + b;
}
```

5. **Functions are called** by using their name followed by parentheses containing any required arguments. When a function is called, control is transferred to the function, and it executes its code.

```
int result = add(3, 4);
```

6. **Variables declared** inside a function are local to that function and cannot be accessed from outside. Similarly, variables declared outside all functions (global variables) can be accessed from any function in the program.

7. C supports **recursion**, which is the ability of a function to call itself. This is useful for solving problems that can be broken down into smaller, similar subproblems.

```
int factorial(int n) {
    if (n == 0 || n == 1)
        return 1;
    else
        return n * factorial(n - 1);
}
```

Storage Class

8.1 STORAGE CLASSES

We have already mentioned that there are two different ways to characterize variables: by *data type*, and by *storage class* (see Sec. 2.6). Data type refers to the type of information represented by a variable, e.g., integer number, floating-point number, character, etc. Storage class refers to the permanence of a variable, and its *scope* within the program, i.e., the portion of the program over which the variable is recognized.

There are four different storage-class specifications in C: *automatic*, *external*, *static* and *register*. They are identified by the keywords `auto`, `extern`, `static`, and `register`, respectively. We will discuss the *automatic*, *external* and *static* storage classes within this chapter. The *register* storage class will be discussed in Sec. 13.1.

EXAMPLE 8.1 Shown below are several typical variable declarations that include the specification of a storage class.

```
auto int a, b, c;

extern float root1, root2;

static int count = 0;

extern char star;
```

8.2 AUTOMATIC VARIABLES

Automatic variables are always declared within a function and are local to the function in which they are declared; that is, their scope is confined to that function. Automatic variables defined in different functions will therefore be independent of one another, even though they may have the same name.

Any variable declared within a function is interpreted as an automatic variable unless a different storage-class specification is shown within the declaration. This includes formal argument declarations. All of the variables in the programming examples encountered earlier in this book have been automatic variables.

An automatic variable does not retain its value once control is transferred out of its defining function. Therefore, any value assigned to an automatic variable within a function will be lost once the function is exited. If the program logic requires that an automatic variable be assigned a particular value each time the function is executed, that value will have to be reset whenever the function is reentered (i.e., whenever the function is accessed).

8.3 EXTERNAL (GLOBAL) VARIABLES

External variables, in contrast to automatic variables, are not confined to single functions. Their scope extends from the point of definition through the remainder of the program. Hence, they usually span two or more functions, and often an entire program. They are often referred to as *global variables.*

Since external variables are recognized globally, they can be accessed from any function that falls within their scope. They retain their assigned values within this scope. Therefore an external variable can be assigned a value within one function, and this value can be used (by accessing the external variable) within another function.

In a single-file program, static variables are defined within individual functions and therefore have the same scope as automatic variables; i.e., they are local to the functions in which they are defined. Unlike automatic variables, however, static variables retain their values throughout the life of the program. Thus, if a function is exited and then re-entered at a later time, the static variables defined within that function will retain their former values. This feature allows functions to retain information permanently throughout the execution of a program.

Arrays

9.1 DEFINING AN ARRAY

Arrays are defined in much the same manner as ordinary variables, except that each array name must be accompanied by a size specification (i.e., the number of elements). For a one-dimensional array, the size is specified by a positive integer expression, enclosed in square brackets. The expression is usually written as a positive integer constant.

In general terms, a one-dimensional array definition may be expressed as

storage-class data-type array[*expression*];

EXAMPLE 9.1 Several typical one-dimensional array definitions are shown below.

```
int x[100];
char text[80];
static char message[25];
static float n[12];
```

9.2 PROCESSING AN ARRAY

Single operations which involve entire arrays are not permitted in C. Thus, if a and b are similar arrays (i.e., same data type, same dimensionality and same size), assignment operations, comparison operations, etc. must be carried out on an element-by-element basis. This is usually accomplished within a loop, where each pass through the loop is used to process one array element. The number of passes through the loop will therefore equal the number of array elements to be processed.

9.3 PASSING ARRAYS TO FUNCTIONS

An entire array can be passed to a function as an argument. The manner in which the array is passed differs markedly, however, from that of an ordinary variable.

To pass an array to a function, the array name must appear by itself, without brackets or subscripts, as an actual argument within the function call. The corresponding formal argument is written in the same manner, though it must be declared as an array within the formal argument declarations. When declaring a one-dimensional array as a formal argument, the array name is written with a pair of empty square brackets. The size of the array is not specified within the formal argument declaration.

9.4 MULTIDIMENSIONAL ARRAYS

Multidimensional arrays are defined in much the same manner as one-dimensional arrays, except that a separate pair of square brackets is required for each subscript. Thus, a two-dimensional array will require two pairs of square brackets, a three-dimensional array will require three pairs of square brackets, and so on.

In general terms, a multidimensional array definition can be written as

```
storage-class  data-type  array[expression 1][expression 2] . . .[expression n];
```

Pointers

Pointers

A *pointer* is a variable that represents the *location* (rather than the *value*) of a data item, such as a variable or an array element. Pointers are used frequently in C, as they have a number of useful applications. For example, pointers can be used to pass information back and forth between a function and its reference point. In particular, pointers provide a way to return multiple data items from a function via function arguments. Pointers also permit references to other functions to be specified as arguments to a given function. This has the effect of passing functions as arguments to the given function.

The data item represented by v (i.e., the data item stored in v's memory cells) can be accessed by the expression *pv, where * is a unary operator, called the *indirection operator*, that operates only on a pointer variable. Therefore, *pv and v both represent the same data item (i.e., the contents of the same memory cells). Furthermore, if we write pv = &v and u = *pv, then u and v will both represent the same value; i.e., the value of v will indirectly be assigned to u. (It is assumed that u and v are of the same data type.)

10.2 POINTER DECLARATIONS

Pointer variables, like all other variables, must be declared before they may be used in a C program. The interpretation of a pointer declaration differs, however, from the interpretation of other variable declarations. When a pointer variable is declared, the variable name must be preceded by an asterisk (*). This identifies the fact that the variable is a pointer. The data type that appears in the declaration refers to the *object* of the pointer, i.e., the data item that is stored in the address represented by the pointer, rather than the pointer itself.

Thus, a pointer declaration may be written in general terms as

```
data-type  *ptvar;
```

where *ptvar* is the name of the pointer variable, and *data-type* refers to the data type of the pointer's object. Remember that an asterisk must precede *ptvar*.

10.3 PASSING POINTERS TO A FUNCTION

Pointers are often passed to a function as arguments. This allows data items within the calling portion of the program to be accessed by the function, altered within the function, and then returned to the calling portion of the program in altered form. We refer to this use of pointers as passing arguments by *reference* (or by *address* or by *location*), in contrast to passing arguments by *value* as discussed in Chap. 7.

10.4 POINTERS AND ONE-DIMENSIONAL ARRAYS

Recall that an array name is really a pointer to the first element in the array. Therefore, if x is a one-dimensional array, then the address of the first array element can be expressed as either &x[0] or simply as x. Moreover, the address of the second array element can be written as either &x[1] or as (x + 1), and so on. In general, the address of array element (i + 1) can be expressed as either &x[i] or as (x + i). Thus we have two different ways to write the address of any array element: We can write the actual array element, preceded by an ampersand; or we can write an expression in which the subscript is added to the array name.

EXAMPLE 10.17 Consider the simple C program shown below.

```
#include <stdio.h>

main()

{
    int *px;          /* pointer to an integer */
    int i = 1;
    float f = 0.3;
    double d = 0.005;
    char c = '*';

    px = &i;
    printf("Values:   i=%i   f=%f   d=%f   c=%c\n\n", i, f, d, c);
    printf("Addresses:   &i=%X   &f=%X   &d=%X   &c=%X\n\n", &i, &f, &d, &c);
    printf("Pointer values:   px=%X   px + 1=%X   px + 2=%X   px + 3=%X",
                                                  px, px + 1, px + 2, px + 3);
}
```

```
Values:   i=1   f=0.300000   d=0.005000   c=*

Addresses:   &i=117E   &f=1180   &d=1186   &c=118E

Pointer values:   px=117E   px + 1=1180   px + 2=1182   px + 3=1184
```

10.7 POINTERS AND MULTIDIMENSIONAL ARRAYS

Since a one-dimensional array can be represented in terms of a pointer (the array name) and an offset (the subscript), it is reasonable to expect that a multidimensional array can also be represented with an equivalent pointer notation. This is indeed the case. A two-dimensional array, for example, is actually a collection of one-dimensional arrays. Therefore, we can define a two-dimensional array as a pointer to a group of contiguous one-dimensional arrays. Thus, a two-dimensional array declaration can be written as

data-type (**ptvar*) [*expression 2*] ;

rather than

data-type *array*[*expression 1*] [*expression 2*] ;

This concept can be generalized to higher-dimensional arrays; that is,

data-type (**ptvar*) [*expression 2*] [*expression 3*] . . . [*expression n*] ;

replaces

data-type *array*[*expression 1*] [*expression 2*] . . . [*expression n*] ;

EXAMPLE 10.21 Suppose x is a two-dimensional integer array having 10 rows and 20 columns, as declared in the previous example. The item in row 2, column 5 can be accessed by writing either

```
x[2][5]
```

or

```
*(*(x + 2) + 5)
```

The second form requires some explanation. First, note that `(x + 2)` is a pointer to row 2. Therefore the object of this pointer, `*(x + 2)`, refers to the entire row. Since row 2 is a one-dimensional array, `*(x + 2)` is actually a pointer to the first element in row 2. We now add 5 to this pointer. Hence, `(*(x + 2) + 5)` is a pointer to element 5 (i.e., the sixth element) in row 2. The object of this pointer, `*(*(x + 2) + 5)`, therefore refers to the item in column 5 of row 2, which is `x[2][5]`. These relationships are illustrated in Fig. 10.5.

10.8 ARRAYS OF POINTERS

A multidimensional array can be expressed in terms of an array of pointers rather than a pointer to a group of contiguous arrays. In such situations the newly defined array will have one less dimension than the original multidimensional array. Each pointer will indicate the beginning of a separate $(n - 1)$-dimensional array.

In general terms, a two-dimensional array can be defined as a one-dimensional array of pointers by writing

data-type **array*[*expression 1*] ;

rather than the conventional array definition,

data-type *array*[*expression 1*] [*expression 2*] ;

Structures and Unions

In Chap. 9 we studied the array, which is a data structure whose elements are all of the same data type. We now turn our attention to the *structure*, in which the individual elements can differ in type. Thus, a single structure might contain integer elements, floating-point elements and character elements. Pointers, arrays and other structures can also be included as elements within a structure. The individual structure elements are referred to as *members*.

This chapter is concerned with the use of structures within a C program. We will see how structures are defined, and how their individual members are accessed and processed within a program. The relationships between structures and pointers, arrays and functions will also be examined.

Closely associated with the structure is the *union*, which also contains multiple members. Unlike a structure, however, the members of a union share the same storage area, even though the individual members may differ in type. Thus, a union permits several different data items to be stored in the same portion of the computer's memory at different times. We will see how unions are defined and utilized within a C program.

11.1 DEFINING A STRUCTURE

Structure declarations are somewhat more complicated than array declarations, since a structure must be defined in terms of its individual members. In general terms, the composition of a structure may be defined as

```
struct  tag  {
        member 1;
        member 2;
        . . . . .
        member m;
};
```

Once the composition of the structure has been defined, individual structure-type variables can be declared as follows:

```
storage-class  struct  tag  variable 1,  variable 2,  . . .,  variable n;
```

where *storage-class* is an optional storage class specifier, `struct` is a required keyword, *tag* is the name that appeared in the structure declaration, and *variable 1*, *variable 2*, . . ., *variable n* are structure variables of type *tag*.

EXAMPLE 11.1 A typical structure declaration is shown below.

```
struct account  {
        int acct_no;
        char acct_type;
        char name[80];
        float balance;
};
```

This structure is named `account` (i.e., the tag is `account`). It contains four members: an integer quantity (`acct_no`), a single character (`acct_type`), an 80-element character array (`name[80]`), and a floating-point quantity (`balance`). The composition of this account is illustrated schematically in Fig. 11.1.

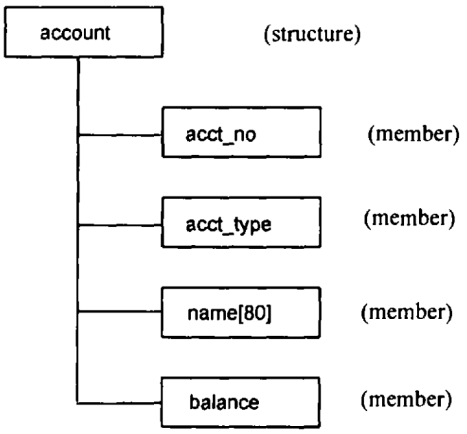

Fig. 11.1

We can now declare the structure variables `oldcustomer` and `newcustomer` as follows.

```
struct  account  oldcustomer, newcustomer;
```

```
struct account  {
     int acct_no;
     char acct_type;
     char name[80];
     float balance;
}  oldcustomer, newcustomer;
```

Thus, `oldcustomer` and `newcustomer` are structure variables of type `account`.

Since the variable declarations are now combined with the declaration of the structure type, the tag (i.e., `account`) need not be included. Thus, the above declaration can also be written as

```
struct  {
     int acct_no;
     char acct_type;
     char name[80];
     float balance;
}  oldcustomer, newcustomer;
```

EXAMPLE 11.4 This example illustrates the assignment of initial values to the members of a structure variable.

```
struct date  {
     int month;
     int day;
     int year;
};
```

```
struct account  {
     int acct_no;
     char acct_type;
     char name[80];
     float balance;
     struct date lastpayment;
};

static  struct  account  customer = {12345, 'R', "John W. Smith", 586.30, 5, 24, 90};
```

11.2 PROCESSING A STRUCTURE

The members of a structure are usually processed individually, as separate entities. Therefore, we must be able to access the individual structure members. A structure member can be accessed by writing

variable.member

where *variable* refers to the name of a structure-type variable, and *member* refers to the name of a member within the structure. Notice the period (.) that separates the variable name from the member name. This period is an operator; it is a member of the highest precedence group, and its associativity is left to right (see Appendix C).

Expression	*Interpretation*
`++customer.balance`	Increment the value of `customer.balance`
`customer.balance++`	Increment the value of `customer.balance` after accessing its value
`--customer.acct_no`	Decrement the value of `customer.acct_no`
`&customer`	Access the beginning address of `customer`
`&customer.acct_no`	Access the address of `customer.acctno`

More complex expressions involving the repeated use of the period operator may also be written. For example, if a structure member is itself a structure, then a member of the embedded structure can be accessed by writing

variable.member.submember

where *member* refers to the name of the member within the outer structure, and *submember* refers to the name of the member within the embedded structure. Similarly, if a structure member is an array, then an individual array element can be accessed by writing

variable.member[*expression*]

where *expression* is a nonnegative value that indicates the array element.

11.4 STRUCTURES AND POINTERS

The beginning address of a structure can be accessed in the same manner as any other address, through the use of the address (&) operator. Thus, if *variable* represents a structure-type variable, then *&variable* represents the starting address of that variable. Moreover, we can declare a pointer variable for a structure by writing

```
type  *ptvar;
```

where *type* is a data type that identifies the composition of the structure, and *ptvar* represents the name of the pointer variable. We can then assign the beginning address of a structure variable to this pointer by writing

```
ptvar = &variable;
```

EXAMPLE 11.19 Consider the following structure declaration, which is a variation of the declaration presented in Example 11.1.

```
typedef struct  {
     int acct_no;
     char acct_type;
     char name[80];
     float balance;
}  account;

account  customer, *pc;
```

In this example `customer` is a structure variable of type `account`, and `pc` is a pointer variable whose object is a structure variable of type `account`. Thus, the beginning address of `customer` can be assigned to `pc` by writing

```
pc = &customer;
```

The variable and pointer declarations can be combined with the structure declaration by writing

```
struct  {
     member 1;
     member 2;
     . . . . .
     member m;
}  variable,  *ptvar;
```

where *variable* again represents a structure-type variable, and *ptvar* represents the name of a pointer variable.

11.5 PASSING STRUCTURES TO FUNCTIONS

There are several different ways to pass structure-type information to or from a function. Structure members can be transferred individually, or entire structures can be transferred. The mechanics for carrying out the transfers vary, depending on the type of transfer (individual members or complete structures) and the particular version of C.

Individual structure members can be passed to a function as arguments in the function call, and a single structure member can be returned via the `return` statement. To do so, each structure member is treated the same as an ordinary single-valued variable.

EXAMPLE 11.25 Consider the simple C program shown below.

```
#include <stdio.h>

typedef struct {
     char *name;
     int acct_no;
     char acct_type;
     float balance;
} record;

main()   /* transfer a structure-type pointer to a function */
{
    void adjust(record *pt);       /* function declaration */

    static record customer = {"Smith", 3333, 'C', 33.33};

    printf("%s %d %c %.2f\n", customer.name, customer.acct_no,
                              customer.acct_type, customer.balance);
    adjust(&customer);
    printf("%s %d %c %.2f\n", customer.name, customer.acct_no,
                              customer.acct_type, customer.balance);
}


void adjust(record *pt)         /* function definition */

{
    pt->name = "Jones";
    pt->acct_no = 9999;
    pt->acct_type = 'R';
    pt->balance = 99.99;
    return;
}
```

This program illustrates the transfer of a structure to a function by passing the structure's address (a pointer) to the function. In particular, `customer` is a static structure of type `record`, whose members are assigned an initial set of values. These initial values are displayed when the program begins to execute. The structure's address is then passed to the function `adjust`, where different values are assigned to the members of the structure.

Within `adjust`, the formal argument declaration defines `pt` as a pointer to a structure of type `record`. Also, notice the empty `return` statement; i.e., nothing is explicitly returned from `adjust` to `main`.

Within `main`, we see that the current values assigned to the members of `customer` are displayed a second time, after `adjust` has been accessed. Thus, the program illustrates whether or not the changes made in `adjust` carry over to the calling portion of the program.

Execution of the program results in the following output.

```
Smith 3333 C 33.33
Jones 9999 R 99.99
```

11.7 UNIONS

Unions, like structures, contain members whose individual data types may differ from one another. However, the members within a union all share the same storage area within the computer's memory, whereas each member within a structure is assigned its own unique storage area. Thus, unions are used to conserve memory. They are useful for applications involving multiple members, where values need not be assigned to all of the members at any one time.

Within a union, the bookkeeping required to store members whose data types are different (having different memory requirements) is handled automatically by the compiler. However, the user must keep track of what type of information is stored at any given time. An attempt to access the wrong type of information will produce meaningless results.

In general terms, the composition of a union may be defined as

```
union  tag  {
    member 1;
    member 2;
    . . . . .
    member m;
};
```

where `union` is a required keyword and the other terms have the same meaning as in a structure definition (see Sec. 11.1). Individual union variables can then be declared as

storage-class union *tag variable 1, variable 2, . . ., variable n*;

where *storage-class* is an optional storage class specifier, `union` is a required keyword, *tag* is the name that appeared in the union definition, and *variable 1, variable 2, . . ., variable n* are union variables of type *tag*.

The two declarations may be combined, just as we did with structures. Thus, we can write

```
storage-class  union  tag  {
          member 1;
          member 2;
          . . . . .
          member m;
}  variable 1,  variable 2,  . . .,  variable n;
```

The *tag* is optional in this type of declaration.

EXAMPLE 11.33 A C program contains the following union declaration.

```
union  id  {
    char color[12];
    int size;
}  shirt, blouse;
```

Here we have two union variables, `shirt` and `blouse`, of type `id`. Each variable can represent either a 12-character string (`color`) or an integer quantity (`size`) at any one time.

The 12-character string will require more storage area within the computer's memory than the integer quantity. Therefore, a block of memory large enough for the 12-character string will be allocated to each union variable. The compiler will automatically distinguish between the 12-character array and the integer quantity within the given block of memory, as required.

A union may be a member of a structure, and a structure may be a member of a union. Moreover, structures and unions may be freely mixed with arrays.

EXAMPLE 11.34 A C program contains the following declarations.

```
union  id  {
    char color[12];
    int size;
};
struct  clothes  {
    char manufacturer[20];
    float cost;
    union id description;
}  shirt, blouse;
```

EXAMPLE 11.36 Shown below is a simple C program that includes the assignment of initial values to a structure variable.

```
#include <stdio.h>

main()

{
    union id  {
        char color[12];
        int size;
    };

    struct clothes {
        char manufacturer[20];
        float cost;
        union id description;
    };

    static struct clothes shirt = {"American", 25.00, "white"};

    printf("%d\n", sizeof(union id));
    printf("%s %5.2f ", shirt.manufacturer, shirt.cost);
    printf("%s %d\n", shirt.description.color, shirt.description.size);

    shirt.description.size = 12;
    printf("%s %5.2f ", shirt.manufacturer, shirt.cost);
    printf("%s %d\n", shirt.description.color, shirt.description.size);
}
```