Keywords

auto - break - case - char - const - continue - default - do - double - else - enum - extern - float - for - goto - if - int - long - register - return - short - signed - sizeof - static - struct - switch - typedef - union - unsigned - void - volatile - while.

* must be written in small case.

Data Types

	Size	Size Qualifiers		Sign Qualifiers	
int	16 or 32 bit (depends on the machine)	**short** short int +16 bit	**long** long int -32 bit or 64 bit	**signed**	**unsigned**
char	8 bit	N/A		**signed**	**unsigned**
float	32 bit	N/A		N/A	
double	64 bit	**long** long double +128 bit		N/A	

void is a type used with
- **functions** to indicate that the function doesn't return a value
  `void funcl(void);`
- **pointers** to indicate a generic pointer that can be assigned to or from other pointers with other data types

`uint8_t, int8_t, uint16_t, uint32_t, ...` are used by some programmers to get rid of confusion of remembering the size of long, short, etc.

C
CrashSheet

Prepared by Electgon
www.electgon.com
ma_ext@gmx.net
Version 4.0
04.01.2020

References:
* Noel Kalicharan, C by example, Cambridge University Press 1994

Structures

Structures are collection of one or more variables

```
struct PersonData{
  char name[20];
  int age;
};
```

Member of a structure cannot be the same as the structure being defined.

In effect, this declares a new type called `struct PersonData`.

We can now declare variables of that type `struct PersonData Person1;`

If we have `struct PersonData human;` we can make `human = Person1;` This will assign all the corresponding fields.

`Person1.name` refers to the first field. `Person1.age` refers to the second field.

`typedef` is used to give a name for some existing types in C.

```
typedef int SpecNum;
SpecNum amount; ≡ int amount;
```

typedef is used usually with structures

```
typedef struct PersonData {
  char name [20];
  int age;
} PersonType;
```

Then we can use `PersonType human;`

Pointer to a structure:

```
PersonType *PersonPtr;
PersonPtr -> age = 25; ≡ human.age = 25;
```

Structures can be passed by value or reference

Assume `void func1 (PersonType InputPerson);` and `void func2(PersonType *PiontedPerson);`
Passing by value: `func1(human);`
Passing by reference: `func2(&human);`

Nested structure: Assume

```
struct fullname {
  char firstname[20];
  char lastname[20];
};
```

Another structure can be defined as

```
struct PersonData{
  struct fullname Name;
  int age;
} human;
```

To access fields of the structure: `human.Name.firstname`

Variables

* To declare a variable, we shall use identifier that begins with a letter or underscore.
* it can contain letters or underscore or digits.
* In ANSI C only first 31 characters are significant.
* Case Sensitive (num is not NUM).
* Single or group declaration are possible:
  `int num;`
  or `int num, count, sum;`

Modifiers

register
* The value of a register variable will be saved in a machine register.
* It is not allowed to guess the address using '&' as the register is not in the memory (i.e. has no address).
* Can be defined for data types int, char, void only.

extern
* default scope for variables defined outside functions.
* If no initialization is defined, C initializes it to 0.

`extern int num;`

auto
* default scope for variables defined in a function.
* Its scope is limited to the function contains it.

`int num; = auto int num;`

const
* Once its value is defined at initialization, it can't be modified later on.

`const int cash limit = 10000;`

* Can be applied to any data type (int, char, etc).

static
* If defined for automatic variable, its value is retained between calls to the function.
* If defined for external variable, its scope will be limited only to the file contains definition of this external variable.
* If no initialization, C initializes it to 0.

volatile
* used for variables that can be changed by external execution. i.e the variable is used in the current program but its value may be changed by another program concurrently

`volatile int num;`

Example:

```
void CountFun() {
  static int count;
  count = count + 1;
}
```

After first call of CountFun count will be 1, After soecnd call count will be 2.

Example *

```
int ch; // this variable can be called in main, fun1, fun2, fun3 without any more declaration.
main () {
... statements; ....
}

int count;  //only fun1, fun2, fun3 can call it

fun1() {
extern int number; // we must use 'extern' as we need to use the external variable
... statements; ...
}

fun2() {
extern int ch; // it is not necessary to delcare again (already declared in first line)
...statements;...
}

int number; // its scope is fun3 and fun1 (declared in fun1 as extern)

fun3() {
... statements;..
}
```

Functions

Factorial.c

```
void GroupFactorial (int Val[], int Len);
int SingleFactorial (int n);

main() {
int j, num[5];
for (j=0; j<5; j++) num[j]=5-j;

GroupFactorial(num,3);
for (j=0; j<3; j++) printf("%d ", num[j]);
}

void GroupFactorial (int Val[], int Len) {
int j;
for (j=0; j<Len; j++) Val[j]=SingleFactorial(Val[j]);
}

int SingleFactorial (int n) {
if(n < 0) return 0;
if(n == 0) return 1;
return (n * SingleFactorial(n-1));
}
```

result is 120 24 6

To use a function, a prototype shall be declared first.

We may write `void GroupFactorial (int Val[], int Len, ...);` if we want to make the function accepts as many arguments as possible. Library stdarg.h is needed then

It is possible to call main with arguments

```
main (int argc, char *argv[]) {
... statements; ...
}
```

`argc`: stores no of arguments including the calling name
`argv[]`: stores each argument in sequence

Arrays are passed by reference (the first element is passed to the function.)

Execution of a function is terminated when the closing brace is encountered or a return statement.

Passing an argument to a function is passing by value. i.e. the original variable is not passed but the value of this variable is passed (not the address).

In C, it is possible for a function to call itself, either directly or indirectly. In the direct case, the body of a function f contains a call to f. In the indirect case, a function f may call g which may call h which in turn calls f.

Operators

Arithmetic

+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Remainder

Assignment

=	a = 10; b = 5;
+=	b += a; ≡ b = b+a;
-=	b -= a; ≡ b = b – a;
*=	b *= a; ≡ b = b * a;
/=	b /= a; ≡ b = b / a;
%=	b %= a; ≡ b = b % a;

Relational

==	Check if both terms are equal
!=	Not equal
>	Greater than
<	Smaller than
<=	Greater than or equal
>=	Smaller than or equal

Logical

&&	And. e.g. check if (a == b) &&(a == c)
||	Or
!	Not

Bit wise

&	And
|	Or
~	Not
^	Exclusive or
<<	Shift left
>>	Shift right

Others

?:	expr1 ? expr2 : expr3; it means if expr1 true execute expr2 other wise execute expr3
++	To increment a variable by 1. e.g. assume a = 10 then a++ will make a=11
--	To decrement by 1
->	Used with pointers to structures
.	Used to get element of a structure
&	Used to get address of a variable

Assume n=7
a=n++; ✔ a=7, n=8
a=++n; ✔ a=8, n=8

Pointers

* If n is an int variable stored in memory, '&n' gives the address of n.
* By assigning `ptr=&n;` ptr will contain the address of n.

`ptr` shall be declared as `int *ptr;`
`*ptr = n  &n = ptr`

* It can be pointer to any type `char, float,` etc.
* It can be also `void *ptr;` to point to any data type.

* We can assign pointer to other pointer `ptr1 = ptr2;`

* An array name is a pointer: `char word [20];`
* `word = &word[0].`

So we can use: `char *errormessage;`
`errormessage = "please provide right num \n";`

And we can change or assign this message:
`oldmessage = errormessage;`
`errormessage = "can't accept negative number \n";`
`errormessage` is the address of `e`, `errormessage+1` is the address of `r`, `errormessage+2` is the address of `r`.

A function name is a pointer to the function.

```
void maketable(int first, int last, float (*f) (int)) {
int j;
for (j=first; j<=last ; j++)
  printf("%d %f \n", j, (*f)(j));
}
```

The heading says that maketable takes three arguments – the first two are integers and the third is a pointer to a function which takes an int argument and returns a float value.

Branching & Looping

```
if (condition) {
...statements;...
}
```

```
if (condition) {
...statements;...
} else {
...statements;...
}
```

```
if (condition) { ...statements;...
} else if (condition){
...statements;...
} else { ...statements;...
}
```

IF

FOR

```
for (expr1; expr2; expr3) {
...statements;...
}
```

```
for (expr1; expr2;)
{statements;}
```

```
for (;expr2;) {statements;}
```

```
do {
...statements;...}
while (condition);
```

DO ... WHILE

GOTO

```
Label : statements;
        ....
goto Label;
```

Not used by programmers, since other controls are sufficient

```
while (condition) {
  ...statements;...
}
```

WHILE

Continue / Break

SWITCH ... CASE

```
switch (expr) {
case 0: statements;  break;
case 1: statements;  break;
default: statements; break;
}
```

`continue` is used to terminate current loop and starts next loop.

`break` is used to terminate execution of next evaluation.

Bit-fields

In declaring a member of a structure, it is permissible to specify the number of bits which the member may occupy.

Called bit-field. By default, they are treated as unsigned integer.

Can be treated as signed integer if specified `signed opcode :5;`

It is a set of adjacent bits within a single storage unit.
If the storage unit uses 16 bits for one int, they will be stored in it.

If name of the field is omitted, it creates hole bits

```
struct machinetype {
  opcode : 5;
  reg: 2;
  address: 9;
} instruction;
```

To assign value for these bit-fields:

```
instruction.opcode = 023;
instruction.reg = 01;
instruction.address = 0257;
```

Bit-fields has no address, so it is illegal to apply & to it.

```
struct {
Field_1: 6;
Field_2: 6;
       : 4;
Field_3: 8;
} abc;
```

This will use two storage units (assuming int is stored in 16 bits)

Arrays

Declaration

Simple declaration: `int count[100];`

Size of the array must be constant

```
int a_size;
int wrongArray[a_size];
```

This is not accepted

Two dimensional array: `int num[2][3]={{13, 15, 17},{21, 23, 25}};`

Initialization

It is possible to initialize some or all elements `int num[3]={17, 18, 19};`

To initialize array of characters →

```
char name[5] = {'a', 'b', 'c'};
char name[5] = "abcde";
char message[]="welcome";
```

In some compilers, initialization is possible only for static or extern arrays.

More

Strings are handled as array of characters `char color [5][10]={"red", "orange", "yellow", "green", "blue"};`

It is optional to specify size of the rows `int num[][3]={{13, 15, 17},{21, 23, 25}};`

Array of structures

```
Struct PersonData {char letter; int age;};
Struct PersonData group [] = {{'a', 25}, {'b', 28}};
```

Unions

* Union is a special data type available in C that allows to store different data types in the same memory location. We can define a union with many members, but only one member can contain a value at any given time.
* Unions provide an efficient way of using the same memory location for multiple purposes.

```
Union AllowedData {
  int IntNum;
  float FloNum;
  char Str[20];
} UniData;
```

To access a member: `UniData.IntNum`

Pointers can be defined: `UniData *UniPtr;`

We can get the address: `UniPtr = &UniData;`

Enumeration

```
enum color {
    red, orange, yellow,
    green, blue
};

enum color shirt;
```

Enumeration is used to declare new data type

These values (red, orange, etc) are treated as integers. i.e. C assigns integer value for each value red = 0 orange = 1 yellow = 2 green = 3 blue = 4

We can change the value of this assignment
`enum color {red =1, blue = 3, green = 7};`

We can manipulate these values mathematically only
`printf("%d \n", shirt);`

```
typdef enum {
  red, orange, yellow,
  green, blue
} color;

color shirt;
```

Preprocessor

#include
`#include` enables the contents of another file to be inserted in the position where the directive appears.

#define
`#define MAXLEN25` will replace any MAXLEN by 25
`#define then` will replace then by nothing, i.e. will delete any then written.

#if defined

```
#if defined (COMPILE)
  //..statements;
#endif
------Or------
#ifdef COMPILE
  //...statements;
#endif
```

```
#if DIFF == 1
  #define MAXSIZE 100
#elif DIFF == 2
  #define MAXSIZE 400
#else
  #define MAXSIZE 1000
#endif
```

Compiler def.
We can pass definition of a preprocessor via the compiler using -D
`$gcc -D COMPILE my_c_code.c -o my_c_code`
`$gcc -D DIFF=2 c_file.c -o c_file`

Macros

```
#define square(n) ((n) * (n))
#define factorial(n, nfac) for (nfac = 1; n > 0; n--) nfac *=n
```

Stringizing

`#define func(x) printf(#x)`
`func(this is example);` will be interpreted as `printf("this is example");`

Token-pasting

```
#define maketemp(n) temp##n = 0
maketemp(3);will be interpreted as temp3 = 0;
maketemp(_4); will be interpreted as temp_4 = 0;
```

Include Guard
To avoid multiple include of a header file from multiple files, use guard

```
#ifndef _HEADER_File_h
#define _HEADER_File_h
  // ....
#endif
```

Var. Args.
Macros can accept variable no. of arguments using `__VA_ARGS__`

```
#define my_printf(...) fprintf(__VA_ARGS__)
```

#undef
`#undef` removes an identifier from the list of defined identifiers. Once an identifier has been undefined, it could subsequently be redefined with another value.