Alexandros Tomaras

Supportive Instruction in C

Symposium
on Innovation
of Computer Science
in Higher Education:

Retraining
of Young
Teaching Staff

Department of Informatics
T.E.I. of Athens

24-27.2.04

```
f()
{
  int x;
  x=2;
}
F()
{
  int x;
  x=-3;
}
```

x declared twice; each x is only known to the code that is within the same block as the variable declaration.

```
f()
{ int x;
  scanf("%d", &x);
  if (x==1)
   {
    char s[80];
    printf("enter name: ");
    gets(s);

    process(s);
   }
}
```

Local variables may be declared within any code block.

```
/* return 1 if c is part of string s;
   Ø otherwise */

is_in(s,c)
 char *s, c;
{
  while (*s)
    if (*s==c)
     return 1;
    else
     s++;

  return Ø;
}
```

s, c: they may be used insid the function as normal local variables; as such, they are also dynamic an are destroyed upon exit from the function.

```
int count;
main()
{
 count=1ØØ;
 f1();
}

f1()
{
 int temp;
 temp=count; f2(); printf("count=%d", count);
} /* count=1ØØ */
f2()
{
 int count;
 for (count=1; count<1Ø; count++)
  putchar('.');
}
```

When f2() references **count**, it references only its local variable, not the global one.

```
/* general */                /* specific */
mul(x, y)                    int x, y;
  int x, y;                  mul()
{return (x*y);}              {return (x*y);}
```

general: can be used to return the product of **any** two integers.

specific: can be used to find **only** the product of the global variables x and y.

A global variable can be declared only once; a local, more than once.

```
int i=7;
main()
{
 printf("%d", i); prog2(); printf("%d\n", i);
} /* prog1.c */
```

```
prog2()                          extern int i;
{                                prog2()
 extern int i;         or        {
 i=12;                            i=12;
} /* prog2.c */                  } /* prog2.c */
```

then: cc prog1.c prog2.c

a.out ⟶ 7   12

extern: tells the compiler that the variable types and names, that follow it, have already been declared elsewhere.

```
main()
{
  int i;
  for (i=0; i<3; ++i) f();
}
f()
{
  int x=0;
  static int y=0;
  printf("%d %d\n", x, y);
  ++x; ++y;
}
```

output

0	0
0	1
0	2

x: local

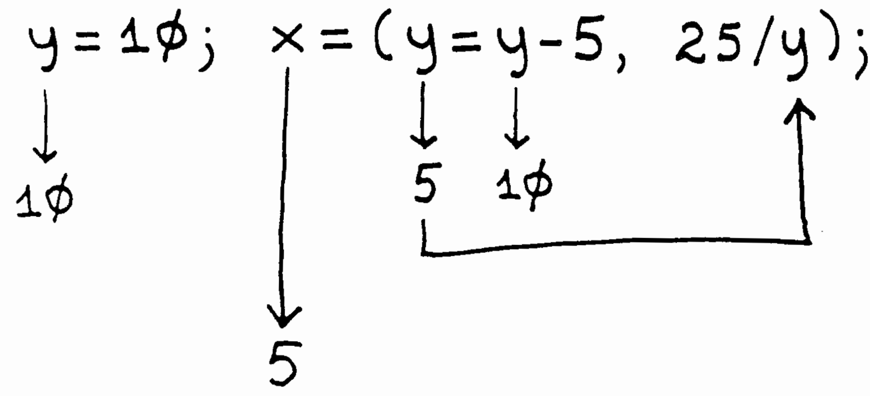
y: static local

```
f(seed)
  int seed;
{
  i=seed;
}
static int i;
F()
{
  i=i+5;
  return(i);
}
```

i: static global; Known only to the file in which it is declared. Routines in other files may not recognize i or alter its content directly.

```
x = (y=3, y+1);
```

y ← 3, x ← 4



```
int_pwr(m, e) /* computes m^e for integers */
 int m;
 register int e;
{
 register int temp;

 temp=1;        → e!=0
 for (; e; e--)
   temp *=m;
 return temp;
}
```

(specifier) register: requests the compiler to keep the value of declared variable in the register of the cpu rather than in memory, where normal variables are stored.

Global *register* variables are disallowed.

```
/* assume: int → 2 bytes, float → 8 bytes */
float f;
printf("%d", sizeof f);
printf(" %d", sizeof(int));
```

output
----
8 2

To compute the size of a type, the type name must be enclosed in ( ). Enclosing variable names in ( ) is not necessary.

→ if x *true* (anything other than 0)

```
if (x) if (y) printf("1"); else printf("2");
```

does:

```
if (x) {if (y) printf("1"); else printf("2");
```

An *else* is linked to the closest *if* that does not already have an *else* statement associated with it.

```
x=10; y= x>9 ? 100 : 200;
```

does the same as:

```
x=10; if (x>9) y=100; else y=200;
```

```
main()
{
 int t; printf(": "); scanf("%d", &t);
 t? f1(t)+f2(t): printf("ø entered");
}   └──────→ t!=ø?
f1(n) int n; {printf("%d ", n);}
f2() {printf("entered");}
```

Any number, other than ø, entered? Then, both f1 and f2 execute.

```
switch(x)
 case 1:
  flag=2;
 case 2:
  flag=3;
  break;
 case 3:
 case 4:
  z=ø; w=7; break;
```

Execution continues into the next case if no *break* statement is present.

case 3 has empty statements. Then, the constants 3, 4 (all two) execute:

z=ø; w=7; break;

```
for (x=ø; x!=123;) scanf("%d", &x);
```

This loop runs until 123 is entered.

---

for (;;) and while(1) create an infinite loop

```
for (;*str==' '; str++);
```

This loop has no body; it skips leading spaces that appear in the stream pointed to by *str*.

```
for (t=ø; t<some_value; t++); /* a time delay loop */
```

```
main()
{
 int t;
 for (prompt(); t=readnum(); prompt())
   sqrnum(t);
}
prompt(){printf(": ");}
readnum(){int t; scanf("%d",&t); return t;}
sqrnum(num)
  int num;
{printf("%d\n", num*num);}
```

Each part of the *for* loop is a function. If th
number entered is ∅, the loop terminates,
because the conditional expression is *false*.

```
f1()
{
 int working=1;
 while (working)
   {
    working=process1();
     if (working)
      working=process2();
     if (working)
      working=process3();
   }
 }
```

Any of the three routines may return *false* and cause the compiler to exit the loop.

```
wait_for_char()
{
  char ch;  ch='\0';
  while (ch!='A')
    ch=getchar();
}
```

A Keyboard input routine that simply loops until the characte A is pressed.

```
pad(s, length)
  char *s;
  int length;
{
  int l;  l=strlen(s);
  while (l<length)
    {
      s[l]=' ';
      l++;
    }
  s[l]='\0';
}
```

Adds spaces to the end of a string to fill the string to a predefined length. I the string is already at t desired length, no spaces are added.

```
while ((ch=getchar())!='A');
```

This code simply loops until A is typed.

```
main()
{
 int t;
 for (t=0; t<100; t++)
   {
    printf("%d  ", t);
    if (t==10)
     break;
   }
}
```

Prints 0 to 10 on the screen.

```
menu()
{
 char ch;
  printf("1. Compose Letter\n");
  printf("2. Encrypt Letter\n");
  printf("3. Quit\n");
  printf("Enter your choice: ");

  do
   {
    ch=getchar(); /* read selection from Keyboard */
    switch(ch)
      {
       case '1':
         compose_letter();
         break;
       case '2':
         encrypt_letter();
         break;
       case '3':
         exit(0); /* return to O.S. */
      }
   } while (ch!='1' && ch!='2' && ch!='3');
}
```

After the display of the options, the program loops until a valid option is selected.

```
for (t=0; t<100; ++t)
  {
   count=1;
   for(;;)
    {
     printf("%d", count);
     count++;
     if (count==10)
      break;
    }
  }
```

Prints 1 through 10 on the screen 100 times.

```
pr_reverse(s)
 char *s;
{
 register int t;
 for (t=strlen(s)-1; t; t--)
   putchar(s[t]);
}
```

Prints a string backwards.

```
display(num)
 int num[ ];
{int i; for (i=0; i<10; i++) printf("%d ",num[i]); }
```

if: int *num; same result occurs

```
print_upper(string)
 char *string;
{
 register int t;

 for (t=0; string[t]; ++t)
   putchar(toupper(string[t]));
}
```

Prints string as uppercase.

```
main() {char s[80]; gets(s); print_upper(s); }
```

formula-1

sizeof(type) * length of array = total bytes

Computes the total size in bytes of a single-dimension array.

formula-2

row * column * sizeof(type) = bytes of memory

Computes the total bytes of memory for a two-dimensional array.

```
main(argc, argv)
  int argc;
  char *argv[ ];
{
  if (argc!=2)
   {
    printf("wrong arguments' number\n");
    exit(ø);
   }
  printf("Hello %s", argv[1]);
}
```

```
A>prog Mary↵
Hello Mary
A>
```

```
prog 1↵
Hello 1
```

argc holds the number of arguments on the command line; it is always at least 1, becau[se] the name of the program qualifies as th[e] first argument. Each element in argv[ ] points to a command line argument. All command line arguments are strings; therefore, any numbers are ASCII numbers and have to be converted by the program into the proper format. argv[ø] points t[o] the first string, which is always the program's name; argv[1] points to the next string and so on.

A> run prog → 2 strings

A> run,prog → 1 string

Only spaces or tabs separate strings.

---

char a[3];	a[Ø]	a[1]	a[2]
element	Ø	1	2
address	1ØØØ	1ØØ1	1ØØ2

How array a appears in memory, if it starts at memory location 1ØØØ.

```
main(){int i[7]; f(i); ...}
```

```
f(str)          f(str)          f(str)
 int *str        int str[7]      int str[ ]
{...}           {...}           {...}
```

Equivalent declarations.

```
char str_array[3Ø][8Ø];
```

3Ø strings, each with a maximum length of 8Ø characters.

```
gets(str_array[2]); ↔ gets(&str_array[2][Ø]);
```

Equivalent statements; call of gets with the third string in str_array as argument.

Consider e.g. the declaration: char p[1Ø];
Then, p and &p[Ø] are identical, that is

```
p==&p[Ø]
```

```
main()
{
 char s1[8Ø], s2[8Ø];
 gets(s1); gets(s2);
 printf("lengths:%d %d\n", strlen(s1), strlen(s2));
 if (!strcmp(s1,s2)) printf("equal strings\n");
 strcat(s1, s2); printf("%s\n", s1);
}
```

```
A>hello hello ↵
  lengths: 5 5
  equal strings
  hellohello
  hello
```

```
int *p, i[1Ø]; p=i; p[5]=7; *(p+5)=7;
```

placement of 7 in the 6th element of i

pointer variables can be indexed as if they were array variables

---

`int a[1Ø][1Ø];`   a, &a[Ø][Ø] are identical

reference, e.g.: a[1][2] or *(a+12)

a[j][k] = *(a+(j*row length)+k)

```
int num[1Ø][1Ø];
...
pr_row(j) int j; {int *p,t; p=&num[j][Ø];
for (t=Ø; t<1Ø; ++t) printf("%d ", *(p+t)); }
```

Prints the contents of the specified row for the global integer array num.

- - - - - - - - - - - - - - - -

Generalization of pr_row follows:

```
pv_row(j, row_dimension, p)
 int j, row_dimension, *p;
{
 int t;
 p=p+(j*row_dimension);
 for (t=0; t<row_dimension; ++t)
   printf("%d ", *(p+t));
}
```

```
char *p; /* creation of dynamic array */
p=malloc(1000); /* get 1000 bytes */
```

p points to the first of 1000 bytes of free memory. A dynamic array uses memory from the heap and is accessed by indexing a pointer to that memory.

A function has a physical location in memory that can be assigned to a pointer. The address assigned to the pointer is the entry point of the function. The pointer can then be used in place of the function's name. The address of a function is obtained by using the function's name without any parentheses or arguments. In C there is no direct way to declare a variable to be a function pointer. Instead, a character pointer is declared and a cast is used to assign the address of the function to it.

```
main(){int strcmp(); char s1[80], s2[80], *p;
p=(char *)strcmp; /* cast */
gets(s1); gets(s2); check(s1, s2, p);}
```

The function 'check' follows:

```
check(a, b, cmp)
 char *a, *b; int (*cmp) ( );
{
 printf("testing for equality\n");
 if (!(*cmp) (a, b))
  printf("equal");
 else
  printf("not equal");
}
```

(*cmp) (a, b) performs the call to the function strcmp( ), which is pointed to by cmp with the arguments a, b.

It is, of course, possible to call check( ) directly: check(s1, s2, strcmp); This call would eliminate the need for an additional pointer variable.

```
char *p;
p=malloc(num_bytes);
free(p);
```

The general form for calling 'malloc' and 'free'.

`char str[6]="hello";` or `char str[ ]="hello";`

This code produces the same result as writing:

```
char str[6]={'h', 'e', 'l', 'l', 'o', '\0'};
```

All strings in C end with a null.

```
#include "malloc.h"
#include "stdio.h"
main()
{
  char *s; register int t;
  s=malloc(80);
  if (!s)
   {
    printf("memory request failed\n");
    exit(1);
   }
  gets(s);
  for (t=strlen(s)-1; t>=0; t--)
   putchar(s[t]);
  free(s);
}
```

Proper way to use a dynamically allocated array to read input from the keyboard using 'gets'. The test on s to ensure that 'malloc' returned a valid pointer is absolutely necessary to prevent accidental use of a null pointer, which would almost certainly cause a system crash.

The program prints the string backwards.

```
#include "malloc.h"
main()
{
 int *p;
 p=(int *)malloc(40*sizeof(int));
 if (!p)
  {
   printf("memory request failed\n";
   exit(1);
  }
 table(p); show(p);
}
table(p) /* compiler: thinks that p is an array */
 int p[4][10];
{
 register int i, j;
 for (j=1; j<11; j++)
   for (i=1; i<5; i++)
     p[i][j]=pwr(j,i);
}
show(p)
 int p[4][10];
{
 register int i, j;
 printf("%10s %10s %10s %10s\n", "N", "N^2", "N^3", "N^4);
 for (j=1; j<11; j++){for (i=1; i<5; i++)
 printf("%10d  ",p[i][j]); printf("\n");}
}
pwr(a,b)
 int a, b;
{
 register int t=1;
 for (;b;b--) t=t*a; return t;
}
```

The output of the program follows:

N	N^2	N^3	N^4
1	1	1	1
2	4	8	16
		⋮	
10	100	1000	10000

We have a 4×10 integer array inside the functions 'show' and 'table'. The difference is that the storage for the array is allocated manually by the malloc-statement rather than automatically by the normal array declaration statement. The use of sizeof guarantees the portability of this program to computers with different-sized integers.

```
int *p;
p++;
p--;
```

If p ← 2000 and integers are 2 bytes long, after p++; p ← 2002. After p--; p ← 1998.

```
p=p+9;
```

Makes p point to the 9th element of p's type beyond the element it currently points to.

You may not multiply or divide pointers. You may not add or subtract two pointers. You may not apply the bitwise shift and mask operators to them. And you may not add or subtract type 'float' or type 'double' to pointers.

```
if (p<q)
 printf("p points to lower memory than q\n");
```

```
char e2[ ] = "write error\n";
printf("%s has length %d\n", e2, sizeof e2);
```

Output: write error has length 13

```
if (!(p=malloc(1ØØ))
 {
  printf("out of memory\n");
  exit(1);
 }
```

Proper way to allocate memory and test for a valid pointer.

```
puts(s)  /* with arrays */
  char *s;
{register int t; for (t=Ø; s[t]; ++t) putchar(s[t]); }

puts(s)  /* with pointers */
  char *s;
{while (*s) putchar(*s++); }
```

```
strcmp(s1, s2)   /* with pointers */
 char *s1, *s2;
{
 while (*s1)
   if (*s1 - *s2)
    return *s1 - *s2;
   else
    {s1++; s2++};
 return '\Ø';
}
```

Returns: Ø, if s1 = s2
<Ø, if s1 < s2
>Ø, if s1 > s2

All strings in C are are terminated by a null, which is a *false* value.

```
int **x[10] /* declaration for an
                  int pointer array; size 10 */
x[2]=&var   /* var's address→3d element
                  of the pointer array */
*x[2] /* to find the value of var */
```

```
display_array(q)
 int *q[ ];
{int t; for (t=0; t<10; t++) printf("%d ", *q[t]);}
```

Passing of an array of pointers into a function.

```
main(){
int x=10, *p, **q;
p=&x; q=&p;
printf("%d", **q);}
```

variable	address	value	
x	1000	10	→ **q
	1001		
p	1002	1000	
	1003		
q	1004	1002	

After a pointer is declared, but before it has been assigned a value, it contains an unknown value. Should you try to use the pointer prior to giving it a value, you will probably crash not only the program, but even your operating system.

```
struct structure_type{
 type variable_name;
        :
}structure_variables;
```

General form of a structure definition.

'structure' is a collection of variables that is referenced under one name. A 'structure definition' forms a template tha may be used to create structure variables The fields that comprise the structure are called 'structure elements'. A structure definition is terminated by a semicolon, because it is a statement.

```
struct addr
{
 char name[3Ø];
 char street[4Ø];
 char city[2Ø];
 unsigned long int zip;
} addr_info;
```

If there is only one structure variable, the structure name is not needed.

```
register int t;
for (t=Ø; addr_info.name[t]; ++t)
 putchar(addr_info.name[t]);
```

Prints the contents of addr_info.name, one character at a time.

```
struct addr addr_info[1ØØ];
```

To declare a 1ØØ-element array of structures of type addr.

```
printf("%d", addr_info[2].zip);
```

To print the zip code of structure 3.

```
struct stru        f1(st.x); passes char
{                            value of x
  char x;
                   f2(st.s); passes address of
  int y;                     string s
  float z;
  char s[10];      f3(st.s[2]); passes char
} st;              value of s[2]
```

```
struct bal
{
  float balance;
  char name[80];
} person;
```

struct bal *p; declaration of a structure pointer

p=&person; address of the structure 'person' into p

(*p). balance; ↔ same as: p→balance;
To reference the element 'balance'.

Programmers call '→': the arrow operator