CS 3411 Systems Programming

Department of Computer Science
Michigan Technological University

Introduction to C
Manual Pages

Usually the most accurate source of information for the system you’re working on

Accessed by the ’man’ command from the terminal, followed by section number, followed by the item you want information on

Sections vary from system to system. You can see this by using the command ’man man’. Commonly, the sections are:

1. User commands
2. System calls
3. Library routines
4. Devices
5. File formats
6. Games
7. Misc.
8. System Administration

The ’info’ command is another option for GNU Software
In a high-level language like C, programmers define variables and operations among variables. Understanding the data types and how they are mapped to memory is crucial to understanding a programming language.

Variables have types and represent objects of various sizes. Some built-in types in C:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>Smallest addressable unit of the machine that can contain basic character set. It is an integer type. Actual type can be either signed or unsigned. Typical size is 8 bits. Signed ([-127, +127]), unsigned ([0, 255]).</td>
</tr>
<tr>
<td>signed char</td>
<td>16 bits in size. Short signed integer type. Signed ([-32,767, +32,767]). Unsigned ([0,65,535]).</td>
</tr>
<tr>
<td>Unsigned char</td>
<td>Under gcc and typical intel code, they are all 32 bits. In other settings, the standard says an int will have at least 16 bits. Signed and unsigned treatment is the same.</td>
</tr>
</tbody>
</table>

Short, short int, signed short, signed short int
High-level languages and the machine

Machine memory is organized like an array where each memory cell is the size of a machine word, say 32 bits:

```
#include <stdio.h>
int x;
char z;
short int a;
int b;
```

A series of declarations will be mapped to (typically) consecutive locations in memory:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Memory Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>4144</td>
</tr>
<tr>
<td>z</td>
<td>4148</td>
</tr>
<tr>
<td>x</td>
<td>4152</td>
</tr>
<tr>
<td>a</td>
<td>4156</td>
</tr>
</tbody>
</table>

```
int main()
{
 ..
}
```

Each variable is allocated one or more words. Any extra bytes left in a word will go unused. The compiler may attempt to minimize such wasted storage by allocating the variables in a different order than they were declared.

We can learn the actual memory location of each variable by taking the address of the variable by using the C language address operator: &.
A Brief Look at Program Execution

<table>
<thead>
<tr>
<th>Text</th>
<th>Data</th>
<th>Heap</th>
<th>Stack</th>
</tr>
</thead>
</table>

Text is executable code. Usually write-protected.
Data is *global* data, both initialized and uninitialized.
Heap is area from which dynamic allocations are made (malloc)
Stack is where function *activation records* pushed/popped.

- Pushed (created) on stack when function invoked, removed on return
- May contain: function parameters, function locals, return address, temporaries, saved state, control link, access link

Usual to preallocate a block of storage for initial heap/stack
Program Execution and Data Locations

```c
int x;
int y = 5;
int a[] = {3, 5, 7, 9};

float foo (int z)
{
    int w; float q;
    // computation here
    return q;
}

int main (){
    char *msg;
    msg = (char *)malloc(1024);
    {
        int abc[1000];
    }
    return 0;
}
```

\( x, y \) are in data segment (\( x \) is uninitialized, \( y \) is initialized).

Code for \( \text{foo, main} \) is in the text segment.

Space for \( w, q \) and \( \text{msg} \) is allocated at the top of the stack when the function is called.

Space for where \( \text{msg} \) is pointing to after the call to \( \text{malloc} \) is on the heap.

The array \( \text{abc} \) will have space allocated on the stack when the block is entered, then popped when the block is exited.
High-level languages and the machine

#include <stdio.h>

int x;
char z;
short int a;
int b;

int main()
{
    int byte_distance;
    int word_distance;
    int some_distance;

    byte_distance = (char *)& x - (char *)& b;
    word_distance = (int *)& x - (int *)& b;
    some_distance = (void *)& x - (void *)& b;

    printf("There are %d bytes between the two declarations \n",byte_distance);
    printf("There are %d words between the two declarations \n",word_distance);
    printf("There are %d units between the two declarations \n",some_distance);
}

Anatomy of a C Program

```
int x;
int y = 5;
int a[] = { 3, 5, 7, 9 };

float foo (int z)
{
    int w; float q;
    // computation here
    return q;
}

char *msg;
int main () {
    char *msg;
    msg = (char *)malloc (1024);

    {
        int abc[1000];
    }
    return 0;
}
```

Let's review the declarations in the code snippet
Anatomy of a C Program

```c
int x;
int y = 5;
int a[] = {3, 5, 7, 9};

float foo(int z)
{
    int w; float q;
    // computation here
    return q;
}

int main()
{
    char *msg;
    msg = (char *)malloc(1024);
    {
        int abc[1000];
    }
    return 0;
}
```

`x`, `y` and `a` are global variables.

`foo` is a declaration for a function that takes one argument, `z`

`w`, `q` and `msg` are function local variables.

`msg` is a pointer. We will discuss them in more detail soon!

What about the array `abc`?
Anatomy of a C Program

Four major types of scope in C

Global scope: Variables in the global scope are accessible at any location that uses this module.

Local scope: Variables in the local scope are only accessible in the functions they’re defined in.

File scope: Variables in file scope are only accessible in the file they are declared in. Indicated by the static keyword.

Block scope: Variables declared within a {, } block. Only accessible within that block.
In C, a function may be used without first being declared.

C does not require explicit prototypes!

The original C specifications assume any undeclared function returns an int and does no checking of its arguments!

This may lead to interesting bugs. Be very careful to ensure functions you’re using are declared unless you know exactly what you’re doing!
If \( x \) is an \textit{int}, then \&\( x \) is the \textit{address of} \( x \).

Pointers are declared like so:

```c
int *px;
px = &x; /* px is a pointer to x */
```

The name of an array of type \( T \) is the address of the first byte of storage for the array, and may be assigned to a pointer of type (\( T \) *).

The \& operator may be applied to variables or array elements.

...But not to expressions!

\& (\( x+1 \)) and \&3 not allowed.
**Review of Pointers in C**

`int *` is like a new type - pointer to `int`.

If part of an expression, `*` is called the *indirection operator*.

```c
int x, y, *px;
px = &x;
y = *px; /* same as y = x; */
```

`*` treats its argument as the address of the target, and accesses that address to fetch the contents.
Pointers can be used in expressions! (*px is okay where x is okay)

- \( y = *px + 1 \); /* \( y = x + 1 \); */
- `printf("%d\n", *px);` /* `printf("%d\n", x);` */
- \( d = \sqrt{(*\text{double})*px} \); /* \( d = \sqrt{(\text{double})*px} \); */
- \( *px = 0; \) /* \( x = 0 \); */
- \( *px += 1; \) /* \( x += 1 \); */
- \( (*px)++; \) /* parantheses necessary */

Pointer Assignments:

```c
int x, *px, *py;
px = &x;
py = px;
*py = 0; /* x = 0 */
```
Assume we have the following declaration:

```c
foo *ptr; /*ptr is addr of a "foo" */
```

What would we get from the following operations?

- `ptr = ptr + 1; /*ptr++; */`
- `ptr = ptr + 6;`
- `*ptr++;`
- `( *ptr )++;`
We can achieve the *effect* of call by reference by passing pointers by *value*.

This gives us the output:

Before : x=3 y=5  
A f t e r : x=5 y=3

Which is what we want!
Examples of Pointer Use: Strings in C

There is no string data type in C. Instead, a string is assumed to be a sequence of `char` terminated by a zero byte.

A `char *` is generally used as a string; just a pointer to the first char in the zero-terminated sequence of chars.

Careful when *declaring* a string:

```c
char *STR; /* Only memory allocated is to pointer variable */
char str[20]; /* 20 bytes allocated to hold contents of string */
```
Some examples of string functions:

```c
#include <stdio.h>
int mystrlen ( s )
char *s;
{
    char *p;
    p = s;
    while ( *p ) p++;
    return p−s;
}

strcpyv 1 ( t , s )
char *s , *t ;
{
    while ( (( *t = *s ) != ' \0' )
        s++;
        t++;  
    }
}

strcpyv 2 ( t , s )
char *s , *t ;
{

```c
while (( *t++ = *s++) != '\0 ');
}

strcpyv 3 ( t , s )
char *s, *t ;
{
    while ( *t ++ = *s ++ ) ;
}

main ( ) {
    char test_str[] = "Hello W_rld ! ";
    char copy_to_str[20];

    printf ( "Original string : %s\n" , test_str );
    printf ( "Length of string : %d\n" , mystrlen ( test_str ) );
    strcpyv 3 ( copy_to_str , test_str );
    printf ( "Copied string : %s\n" , copy_to_str );
    printf ( "Length of string : %d\n" , mystrlen ( copy_to_str ) );
}
```
C Standard I/O

Manual pages available for specific functions:

- `man 3 stdio (An overview)`
- `man 3 printf (Formatted output)`
- `man 3 scanf (Formatted input)`
- `man 3 getc (Character-based input macros)`
- `man 3 putc (Character-based output macros)`

Default I/O Streams: stdin, stdout, stderr

Anything you open with the fopen function is also a stream.

All streams are of the `(FILE *) data type.`
stdio requires a string which defines a format to be used

```c
#include <stdio.h>

main ( ){
  float x; int y;
  char *str; x
  = 3.1;
  y = −20;
  str = "Characters ";
  printf("%.2f%d%s\n", x, y, str);
}
```
In C, we need to pass a pointer argument to scanf to get back values

```c
#include <stdio.h>

main ( ) {
    double sum = 0;
    int val , num = 0;
    while ( scanf ( "%d" , & val ) == 1 ){
        num++;
        sum += ( double ) val ;
    }

    printf ( "Mean is %f \n" , sum/( double )num);
}
```
Memory Allocation in C

No new keyword in C!
Memory allocation is done through `malloc`.
Memory is not managed in C, all allocated memory must be freed!
Freeing memory is done through `free`.
’man 3 malloc’ for more details!
Malloc Example I

/* bintree.c */
#include <malloc.h>
#define NILNODE (struct node *)0

struct node {
  char data;
  struct node *left, *right;
};

main () {
  struct node *gimme (), *n1, *n2, *n3, *n4, *n5, *n6, *n7;
  void inorder ();

  n1 = gimme ('a', NILNODE, NILNODE);
  n2 = gimme ('b', NILNODE, NILNODE);
  n3 = gimme ('c', n1, n2);
  n4 = gimme ('d', NILNODE, NILNODE);
  n5 = gimme ('e', n3, n4);
  n6 = gimme ('f', NILNODE, NILNODE);
  n7 = gimme ('g', n5, n6);
  inorder(n7);
  printf("\n");
}
malloc Example II

```c
struct node *gimme(val, l, r)
char val;
struct node *l, *r;
{
    struct node *tmp;
    tmp = (struct node *)malloc(sizeof(struct node));
    tmp->data = val;
    tmp->left = l;
    tmp->right = r;
    return (tmp);
}

void inorder(r)
struct node *r;
{
    if (r != NILNODE) {
        inorder(r->left);
        printf("%c", r->data);
        inorder(r->right);
    }
}
```
Problems to Avoid

It is always important to keep system programs as bug-free as possible.

Errant programs running in privileged mode can:
- Access/modify system configuration files
- Erase user data
- Halt the system
- And so on!
Buffer Overflow

Writing beyond allocated array bounds

```c
int getUserData () {
    char copy [60];
    ...
    /* User can input string of ANY length */
    gets (buf);
    ...
    /* Copies until string termination in buf */
    strcpy (copy, buf);
}

main () {
    ...
    char input[50];
    char *strPtr;
    ...
    getUserData ();
    /* No string memory allocation for strPtr */
    strcpy (strPtr, input);
}
```
Memory Leak

Losing access to allocated memory segment - We can’t reclaim it!

```c
int func ()
{
    void *ptr;
    /*When function returns, value of ptr inaccessible */
    ptr = malloc (100);
}

main ( ) {
    char *bptr;

    for (i = 1; i <10; i ++) {
        /*Previous ptr value overwritten each iteration */
        bptr = malloc (sizeof (char));
        *bptr = i;
    }
}
```
Dereference Invalid Pointer

```c
int func ( node *n ) {
    if ( n->value == 10) free ( n );
    return ( 0 );
}

main ( ) {
    node *p, *q;
    p = malloc ( sizeof ( node ));
    p->value = 10;
    printf ( "Node p value <%d>" , p->value );
    func ( p );
    /*p has already been freed */
    printf ( "After func p value <%d>
" , p->value );
    /*q was never initialized */
    printf ( "Node q value <%d>
" , q->value );
}
```