Lecture 02: Linux and C Programming Language

CSCE 569 Parallel Computing

Department of Computer Science and Engineering
Yonghong Yan
yanyh@cse.sc.edu
http://cse.sc.edu/~yanyh
Contents

• Remote Login using SSH
• Linux
• C Programming
• Compiling and Linking
• Assignment 1
Computation Server

In the cold and dark server room!

Run Linux/Unix Operating System
Client/Server and SSH (Secure Shell)
Machine for Development for OpenMP and MPI

• Linux machines in Swearingen 1D39 and 3D22
  – All CSCE students by default have access to these machine using their standard login credentials
  • Let me know if you, CSCE or not, cannot access
  – Remote access is also available via SSH over port 222. Naming schema is as follows:
    • l-1d39-01.cse.sc.edu through l-1d39-26.cse.sc.edu
    • l-3d22-01.cse.sc.edu through l-3d22-20.cse.sc.edu

• Restricted to 2GB of data in their home folder (~/).
  – For more space, create a directory in /scratch on the login machine, however that data is not shared and it will only be available on that specific machine.
Putty SSH Connection on Windows
SSH Connection from Linux/Mac OS X Terminal

```
yanh@cocsce-l1d39-15:$/opencv$ exit
logout
Connection to l-1d39-15.cse.sc.edu closed.
MacBook-Pro-7:yanh yanh$ ssh -p 222 l-1d39-15.cse.sc.edu -lyanh -X

Password:
/usr/bin/xauth: file /acct/yanh/.Xauthority does not exist
Duo two-factor login for yanh

Enter a passcode or select one of the following options:

1. Duo Push to XXX-XXX-5878
2. Phone call to XXX-XXX-5878
3. SMS passcodes to XXX-XXX-5878

Passcode or option (1-3): 1

Pushed a login request to your device...
Success. Logging you in...
yanh@cocsce-l1d39-15:$ ls
```

-X for enabling X-windows forwarding so you can use the graphics display on your computer. For Mac OS X, you need have X server software installed, e.g. Xquartz(https://www.xquartz.org/) is the one I use.
It is all about dealing with files and folders

Linux folder: /acct/yanyh/...

- **ls** (list files in the current folder)
  - `$ ls -l`
  - `$ ls -a`
  - `$ ls -la`
  - `$ ls -l --sort=time`
  - `$ ls -l --sort=size -r`
- **cd** (change directory to)
  - `$ cd /usr/bin`
- **pwd** (show current folder name)
  - `$ pwd`
- **~** (home folder)
  - `$ cd ~`
- **~user** (home folder of a user)
  - `$ cd ~weesan`
- **What will “cd ~/weesan” do?**

- **rm** (remove a filer/folder)
  - `$ rm foo`
  - `$ rm -rf foo`
  - `$ rm -i foo`
  - `$ rm -- -foo`
- **cat** (print the file contents to terminal)
  - `$ cat /etc/motd`
  - `$ cat /proc/cpuinfo`
- **cp** (create a copy of a file/folder)
  - `$ cp foo bar`
  - `$ cp -a foo bar`
- **mv** (move a file/folder to another location. Used also for renaming)
  - `$ mv foo bar`
- **mkdir** (create a folder)
  - `$ mkdir foo`
Basic Commands (cont)

- df (Disk usage)
  - $ df -h /
  - $ du -sxh ~/
- man (manual)
  - $ man ls
  - $ man 2 mkdir
  - $ man man
  - $ man -k mkdir
- Manpage sections
  - 1 User-level cmds and apps
    - /bin/mkdir

Smarty:
1. [Tab] key: auto-complete the command sequence
2. ↑ key: to find previous command
3. [Ctl]+r key: to search previous command
Editing a File: Vi/Vim

• 2 modes
  – Input mode
    • ESC to back to cmd mode
  – Command mode
    • Cursor movement
      – h (left), j (down), k (up), l (right)
      – ^f (page down)
      – ^b (page up)
      – ^ (first char.)
      – $ (last char.)
      – G (bottom page)
      – :1 (goto first line)
    • Swtch to input mode
      – a (append)
      – i (insert)
      – o (insert line after)
      – O (insert line before)
  • Delete
    – dd (delete a line)
    – d10d (delete 10 lines)
    – d$ (delete till end of line)
    – dG (delete till end of file)
    – x (current char.)
  • Paste
    – p (paste after)
    – P (paste before)
  • Undo
    – u
  • Search
    – /
  • Save/Quit
    – :w (write)
    – :q (quit)
    – :wq (write and quit)
    – :q! (give up changes)
C Hello World

- **vi hello.c**
- **Switch to editing mode:** `i` or `a`
- **Switching to control mode:** `ESC`
- **Save a file:** in control mode, `:w`
- **To quit,** in control mode, `:q`
- **To quit without saving,** `:q!`
- **Copy/paste a line:** in control model, "yy" and then "p", both from the current cursor
  - 5 line: `5yy` and then `p`
- **To delete a whole line,** in control mode, `: dd`

```c
#include <stdio.h>

/* The simplest C Program */

int main(int argc, char **argv) {
    printf("Hello World\n");
    return 0;
}
```
C Syntax and Hello World

```c
#include <stdio.h>
/* The simplest C Program */
int main(int argc, char **argv)
{
    printf("Hello World!\n");
    return 0;
}
```

- **What do the `< >` mean?**
  - `#include` inserts another file. `.h` files are called "header" files. They contain declarations/definitions needed to interface to libraries and code in other `.c` files.
- **A comment, ignored by the compiler**
  - Blocks of code ("lexical scopes") are marked by `{ ... }
- **The main() function is always where your program starts running.**
- **Return '0' from this function**
Compiling/Building Process in C to Generate Executables

• Compiling/Building process: gcc hello.c –o hello
  – Constructing an executable image for an application
  – FOUR stages
  – Command:
    gcc <options> <source_file.c>

• Compiler Tool
  – gcc (GNU Compiler)
    • man gcc (on Linux m/c)
  – icc (Intel C compiler)
4 Stages of Compiling Process

1. Preprocessing (Those with # ...)
   - Expansion of Header files (#include ... )
   - Substitute macros and inline functions (#define ...)

2. Compilation (the most important one)
   - Generates assembly language
   - Verification of functions usage using prototypes
   - Header files: Prototypes declaration (-l option to provide header folder)

3. Assembling
   - Generates re-locatable object file (contains m/c instructions)
   - nm app.o: To list functions/symbols provided by a an object file
     0000000000000000 T main
     U puts
   - objdump to view object files and disassembly
4 Stages of Compiling Process (contd..)

4. Linking
   - Generates executable file (nm tool used to view exe file)
   - Binds appropriate libraries
     - Static Linking
     - Dynamic Linking (default)

- Loading and Execution (of an executable file)
  - Evaluate size of code and data segment
  - Allocates address space in the user mode and transfers them into memory
  - Load dependent libraries needed by program and links them
  - Invokes Process Manager → Program registration
4 Stages of Compiling Process

Preprocessing
gcc –E hello.c –o hello.i
hello.c → hello.i

Compilation (after preprocessing)
gcc –S hello.i –o hello.s

Assembling (after compilation)
gcc –c hello.s –o hello.o

Linking object files
gcc hello.o –o hello

Output → Executable (a.out)
Run → ./hello (Loader)
Compiling a C Program

- `gcc <options> program_name.c`

- **Options:**

  ---------

  - **-Wall:** Shows all warnings
  - **-o output_file_name:** By default, a.out executable file is created when we compile our program with gcc. Instead, we can specify the output file name using "-o" option.
  - **-g:** Include debugging information in the binary.

- `man gcc`
Linking multiple files to make executable file

• Two programs, prog1.c and prog2.c for one single task
  – To make single executable file using following instructions

  **First**, compile these two files with option "-c"
  gcc -c prog1.c
  gcc -c prog2.c

  **-c**: Tells gcc to compile and assemble the code, but not link.

  We get two files as output, prog1.o and prog2.o
  **Then**, we can link these object files into single executable file
  using below instruction.

  gcc -o prog prog1.o prog2.o

  Now, the output is prog executable file.
  We can run our program using
  ```
  ./prog
  ```
Linking with other libraries

• Normally, compiler will read/link libraries from /usr/lib directory to our program during compilation process.
  – Library are precompiled object files

• To link our programs with libraries like pthreads and realtime libraries (rt library).
  – gcc <options> program_name.c -lpthread -lrt

    -lpthread: Link with pthread library → libpthread.so file
    -lrt: Link with rt library → librt.so file
    Option here is "-l<library>"

Another option "-L<dir>" used to tell gcc compiler search for library file in given <dir> directory.
Compilation, Linking, Execution of C/C++ Programs

usually performed by a compiler, usually in one uninterrupted sequence

http://www.tenouk.com/ModuleW.html
Three Useful Commands

• **nm**: e.g. "nm a.out", "nm libc.so"
  – list **symbols** from object files
    • Symbols: function name, global variables that are exposed or reference by an object file.

• **ldd**: e.g. "ldd a.out", or "ldd hello"
  – List the name and the path of the dynamic library needed by a program

  – **LD_LIBRARY_PATH**: env for setting runtime lib path
    • export LD_LIBRARY_PATH=/acct/yanyh/usr/lib:$LD_LIBRARY_PATH

• **objdump**: objdump –d a.out
  – dump information about object files, including disassembly
sum.c

• Download the file:
  – wget https://passlab.github.io/CSCE569/resources/sum.c
• gcc sum.c –o sum
• ./sum 102400

• vi sum.c
• ldd sum
• nm sum

• Other system commands:
  – cat /proc/cpuinfo to show the CPU and #cores
  – top command to show system usage and memory

Or step by step

  gcc -E sum.c -o sum.i
  gcc -S sum.i -o sum.s
  gcc -c sum.c -o sum.o
  gcc sum.o -o sum
Makefile

sum (exe)

main.o

main.c

sum.h

sum.o

sum.c

sum.h
Makefile

sum: main.o sum.o
    gcc -o sum main.o sum.o

main.o: main.c sum.h
    gcc -c main.c

sum.o: sum.c sum.h
    gcc -c sum.c
Rule syntax

main.o: main.c sum.h

gcc -c main.c

dependency action
cmake: Makefile/Build-System Generator

- Provides single-sourcing for build systems
- Knowledge of many platforms and tools
- Users configure builds through a GUI
Sequential Memory Regions vs Multi-dimensional Array

- Memory is a sequentially accessed using the address of each byte/word
Vector/Matrix and Array in C

• C has row-major storage for multiple dimensional array
  – A[2,2] is followed by A[2,3]

• 3-dimensional array
  – B[3][100][100]

Memory address of A[2][3]
= A[0][0] + offset
= A + sizeof (char) * (2 * # columns + 3)
= 0 + 1 * (2 * 4 + 3) = 11
Store Array in Memory in Row Major

3x4 matrix

<table>
<thead>
<tr>
<th>8</th>
<th>6</th>
<th>5</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

C array for matrix:
int A[3][4]

C storage type
Address of element A[1][2]?

= A + sizeof (int) * (1 * 4 + 2)
= A + 4 * 6 = A + 24
Store Array in Memory in Column Major

3x4 matrix

<table>
<thead>
<tr>
<th></th>
<th>8</th>
<th>6</th>
<th>5</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>9</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>4</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

Column-Major (Column Wise Arrangement)
For a Memory Region to Store Data for an Array in Either Row or Col Major

Row-Major (Row Wise Arrangement)

Column-Major (Column Wise Arrangement)
C Programming
Every Variable is Defined within some scope. A Variable cannot be referenced by name (a.k.a. Symbol) from outside of that scope.

Lexical scopes are defined with curly braces { }.

The scope of Function Arguments is the complete body of that function.

The scope of Variables defined inside a function starts at the definition and ends at the closing brace of the containing block.

The scope of Variables defined outside a function starts at the definition and ends at the end of the file. Called “Global” Vars.

```
void p(char x)
{
    char y; /* p,x */
    char z; /* p,x,y */
    char z; /* p,x,y,z */
}
    /* p */
char z; /* p,z */

void q(char a)
{
    char b; /* p,z,q,a,b */
    {
        char c; /* p,z,q,a,b,c */
    }
    /* p,z,q,a,b,c */
    char d; /* p,z,q,a,b,d (not c) */
} /* p,z,q */
```
Comparison and Mathematical Operators

== equal to
< less than
<= less than or equal
> greater than
>= greater than or equal
!= not equal
&& logical and
|| logical or
! logical not
+
-
*
/
%
& bitwise and
| bitwise or
^ bitwise xor
~ bitwise not
<< shift left
>> shift right

Beware division:
• $17/5 = 3, 17 \% 5 = 2$
• $5 / 10 = 0$ whereas $5 / 10.0 = 0.5$
• Division by 0 will cause a FPE (Float-point exception)

Don’t confuse & and &&..:
$1 \& 2 = 0$ whereas $1 \&\& 2 = <true>$

The rules of precedence are clearly defined but often difficult to remember or non-intuitive. When in doubt, add parentheses to make it explicit.
Assignment Operators

\[ x = y \text{ assign } y \text{ to } x \]
\[ x++ \text{ post-increment } x \]
\[ ++x \text{ pre-increment } x \]
\[ x-- \text{ post-decrement } x \]
\[ --x \text{ pre-decrement } x \]

\[ x += y \text{ assign } (x+y) \text{ to } x \]
\[ x -= y \text{ assign } (x-y) \text{ to } x \]
\[ x *= y \text{ assign } (x*y) \text{ to } x \]
\[ x /= y \text{ assign } (x/y) \text{ to } x \]
\[ x %= y \text{ assign } (x\%y) \text{ to } x \]

Note the difference between \( ++x \) and \( x++ \) (high vs low priority (precedence)):

```c
int x=5;
int y;
y = ++x;
/* x == 6, y == 6 */
```

```c
int x=5;
int y;
y = x++;
/* x == 6, y == 5 */
```

Don’t confuse “=” and “==”!

```c
int x=5;
if (x==6) /* false */
{
    /* ... */
}
/* x is still 5 */
```

```c
int x=5;
if (x=6) /* always true */
{
    /* x is now 6 */
}
/* ... */
```
A Quick Digression About the Compiler

Compilation occurs in two steps: “Preprocessing” and “Compiling”

In Preprocessing, source code is “expanded” into a larger form that is simpler for the compiler to understand. Any line that starts with ‘#' is a line that is interpreted by the Preprocessor.

- Include files are “pasted in” (#include)
- Macros are “expanded” (#define)
- Comments are stripped out ( /* */ , // )
- Continued lines are joined ( \ )

The compiler then converts the resulting text (called translation unit) into binary code the CPU can execute.
C Memory Pointers

• To discuss memory pointers, we need to talk a bit about the concept of memory

• We’ll conclude by touching on a couple of other C elements:
  – *Arrays, typedef, and structs*
The “memory”

Memory: similar to a big table of numbered slots where bytes of data are stored.

The number of a slot is its **Address**. One byte **Value** can be stored in each slot.

Some data values span more than one slot, like the character string “Hello\n”

A **Type** provides a logical meaning to a span of memory. Some simple types are:

<table>
<thead>
<tr>
<th>Addr</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>‘H’ (72)</td>
</tr>
<tr>
<td>5</td>
<td>‘e’ (101)</td>
</tr>
<tr>
<td>6</td>
<td>‘l’ (108)</td>
</tr>
<tr>
<td>7</td>
<td>‘l’ (108)</td>
</tr>
<tr>
<td>8</td>
<td>‘o’ (111)</td>
</tr>
<tr>
<td>9</td>
<td>‘\n’ (10)</td>
</tr>
<tr>
<td>10</td>
<td>‘\0’ (0)</td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

- **char**: a single character (1 slot)
- **char [10]**: an array of 10 characters
- **int**: signed 4 byte integer
- **float**: 4 byte floating point
- **int64_t**: signed 8 byte integer
What is a Variable?

A Variable names a place in memory where you store a Value of a certain Type.

You first Declare a variable by giving it a name and specifying its type and optionally an initial value.

Variable x declared but undefined

The compiler puts x and y somewhere in memory.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Addr</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>4</td>
<td>Some garbage</td>
</tr>
<tr>
<td>y</td>
<td>5</td>
<td>'e' (101)</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>9</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11</td>
<td></td>
</tr>
<tr>
<td></td>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>
Multi-byte Variables

Different types require different amounts of memory. Most architectures store data on “word boundaries”, or even multiples of the size of a primitive data type (int, char).

```c
char x;
char y = 'e';
int z = 0x01020304;
```

0x means the constant is written in hex

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Addr</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>4</td>
<td>Some garbage</td>
</tr>
<tr>
<td>y</td>
<td>5</td>
<td>‘e’ (101)</td>
</tr>
<tr>
<td>z</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

An int requires 4 bytes

padding
Memory, a more detailed view...

- A sequential list of words, starting from 0.
- On 32bit architectures (e.g. Win32): each word is 4 bytes.
- Local variables are stored in the stack.
- Dynamically allocated memory is set aside on the heap (more on this later...)
- For multiple-byte variables, the address is that of the smallest byte (little endian).
What is the value of:
- sizeC
- sizeD
- sizeDarr

**NOTE:** `sizeof` is a compile-time operator that returns the size, in multiples of the size of `char`, of the variable or parenthesized type-specifier that it precedes.
Can a C function modify its arguments?

What if we wanted to implement a function `pow_assign()` that modified its argument, so that these are equivalent:

```c
float p = 2.0;
/* p is 2.0 here */
p = pow(p, 5);
/* p is 32.0 here */
```

```c
float p = 2.0;
/* p is 2.0 here */
pow_assign(p, 5);
/* Is p is 32.0 here? */
```

```c
void pow_assign(float x, uint exp)
{
    float result=1.0;
    int i;
    for (i=0; (i < exp); i++) {
        result = result * x;
    }
    x = result;
}
```

Would this work?

Native function, to use you need `#include <math.h>`
In C you can’t change the value of any variable passed as an argument in a function call…

Pass by value

```c
void pow_assign(float x, uint exp)
{
    float result=1.0;
    int i;
    for (i=0; (i < exp); i++) {
        result = result * x;
    }
    x = result;
}

// a code snippet that uses above function
{
    float p=2.0;
    pow_assign(p, 5);
    // the value of p is 2 here...
}
```

In C, all arguments are passed by value

Keep in mind: pass by value requires the variable to be copied. That copy is then passed to the function. Sometime generating a copy can be expensive…

But, what if the argument is the address of a variable?
C Pointers

• What is a pointer?
  – A variable that contains the memory address of another variable or of a function

• In general, it is safe to assume that on 32 bit architectures pointers occupy one word
  – Pointers to int, char, float, void, etc. (“int*”, “char*”, “*float”, “void*”), they all occupy 4 bytes (one word).

• Pointers: *very* many bugs in C programs are traced back to mishandling of pointers...
Pointers (cont.)

• The need for pointers

  – Needed when you want to modify a variable (its value) inside a function
    • The pointer is passed to that function as an argument

  – Passing large objects to functions without the overhead of copying them first

  – Accessing memory allocated on the heap

  – Referring to functions, i.e. function pointers
A **Valid** pointer is one that points to memory that your program controls. Using invalid pointers will cause non-deterministic behavior

• Very often the code will crash with a SEGV, that is, Segment Violation, or Segmentation Fault.

There are two general causes for these errors:

• Coding errors that end up setting the pointer to a strange number
• Use of a pointer that was at one time valid, but later became invalid

Good practice:

• Initialize pointers to 0 (or NULL). NULL is never a valid pointer value, but it is known to be invalid and means “no pointer set”.

```c
char * get_pointer()
{
    char x=0;
    return &x;
}

{
    char * ptr = get_pointer();
    *ptr = 12; /* valid? */
```
A pointer to a variable allocated on the stack becomes invalid when that variable goes out of scope and the stack frame is “popped”. The pointer will point to an area of the memory that may later get reused and rewritten.

```c
char * get_pointer()
{
    char x=0;
    return &x;
}

int main()
{
    char * ptr = get_pointer();
    *ptr = 12;  /* valid? */
    other_function();
    return 0;
}
```

But now, `ptr` points to a location that’s no longer in use, and will be reused the next time a function is called!

Here is what I get in DevStudio when compiling:
main.cpp(6) : warning C4172: returning address of local variable or temporary
Example: What gets printed out?

```c
int main() {
    int d;
    char c;
    short s;
    int* p;
    int arr[2];
    printf( "%p, %p, %p, %p, %p\n", &d, &c, &s, &p, arr );
    return 0;
}
```

• NOTE: Here &d = 920 (in practice a 4-byte hex number such as 0x22FC3A08)
Example:
Usage of Pointers & Pointer Arithmetic

```c
int main() {
    int d;
    char c;
    short s;
    int* p;
    int arr[2];
    p = &d;
    *p = 10;
    c = (char)1;
    p = arr;
    *(p+1) = 5;
    p[0] = d;
    *( (char*)p + 1 ) = c;
    return 0;
}
```

Q: What are the values stored in arr? [assume little endian architecture]
p = &d;
*p = 10;
c = (char)1;

p = arr;
*(p+1) = 5; // int* p;
p[0] = d;
*( (char*)p + 1 ) = c;

Question: arr[0] = ?
Use of pointers, another example...

- Pass pointer parameters into function

```c
void swap(int *px, int *py)
{
    int temp;
    temp = *px;
    *px = *py;
    *py = temp;
}

int a = 5;
int b = 6;
swap(&a, &b);
```

- What will happen here?

```c
int * a;
int * b;
swap(a, b);
```
Dynamic Memory Allocation (on the Heap)

- Allows the program to determine how much memory it needs *at run time* and to allocate exactly the right amount of storage.
  - It is your responsibility to clean up after you (free the dynamic memory you allocated)

- The region of memory where dynamic allocation and deallocation of memory can take place is called the heap.
Recall Discussion on Dynamic Memory Allocation

Recall that variables are allocated **statically** by having declared with a given size. This allocates them in the stack.

Allocating memory at run-time requires **dynamic** allocation. This allocates them on the heap.

```c
int * alloc_ints(size_t requested_count) {
    int * big_array;
    big_array = (int *)calloc(requested_count, sizeof(int));
    if (big_array == NULL) {
        printf("can't allocate %d ints: %m\n", requested_count);
        return NULL;
    }

    /* big_array[0] through big_array[requested_count-1] are valid and zeroed. */
    return big_array;
}
```

`calloc()` allocates memory for N elements of size k

`sizeof()` reports the size of a type in bytes

Returns NULL if can’t alloc

It’s OK to return this pointer. It will remain valid until it is freed with `free()`. However, it’s a bad practice to return it (if you need is somewhere else, declare and define it there…)
Caveats with Dynamic Memory

Dynamic memory is useful. But it has several caveats:

Whereas the stack is automatically reclaimed, dynamic allocations must be tracked and free()ed when they are no longer needed. With every allocation, be sure to plan how that memory will get freed. Losing track of memory causes “memory leak”.

Whereas the compiler enforces that reclaimed stack space can no longer be reached, it is easy to accidentally keep a pointer to dynamic memory that was freed. Whenever you free memory you must be certain that you will not try to use it again.

Because dynamic memory always uses pointers, there is generally no way for the compiler to statically verify usage of dynamic memory. This means that errors that are detectable with static allocation are not with dynamic
Data Structures

• A data structure is a collection of one or more variables, possibly of different types.

• An example of student record

```c
struct StudRecord {
    char name[50];
    int id;
    int age;
    int major;
};
```
Data Structures (cont.)

• A data structure is also a data type

```c
struct StudRecord my_record;
struct StudRecord * pointer;
pointer = & my_record;
```

• Accessing a field inside a data structure

```c
my_record.id = 10;
// or
pointer->id = 10;
```
Data Structures  (cont.)

• Allocating a data structure instance

  This is a new type now

```c
struct StudRecord* pStudentRecord;
pStudentRecord = (StudRecord*)malloc(sizeof(struct StudRecord));
pStudentRecord ->id = 10;
```

• IMPORTANT:
  – Never calculate the size of a data structure yourself. Rely on the `sizeof()` function
  – Example: Because of memory padding, the size of “struct StudRecord” is 64 (instead of 62 as one might estimate)
typedef Construct

struct StudRecord {
    char name[50];
    int id;
    int age;
    int major;
};

typedef struct StudRecord RECORD_t;

int main() {
    RECORD_t my_record;
    strcpy_s(my_record.name, "Joe Doe");
    my_record.age = 20;
    my_record.id = 6114;
    
    RECORD_t* p = &my_record;
    p->major = 643;
    return 0;
}
Arrays

Arrays in C are composed of a particular type, laid out in memory in a repeating pattern. Array elements are accessed by stepping forward in memory from the base of the array by a multiple of the element size.

/* define an array of 10 chars */
char x[5] = {'t','e','s','t','\0'};

/* access element 0, change its value */
x[0] = 'T';

/* pointer arithmetic to get elt 3 */
char elt3 = *(x+3); /* x[3] */

/* x[0] evaluates to the first element; */
/* x evaluates to the address of the */
/* first element, or &(x[0]) */

/* 0-indexed for loop idiom */
#define COUNT 10
char y[COUNT];
int i;
for (i=0; i<COUNT; i++) {
    /* process y[i] */
    printf("%c\n", y[i]);
}

Brackets specify the count of elements. Initial values optionally set in braces.

Arrays in C are 0-indexed (here, 0…4)

x[3] == *(x+3) == ‘t’ (notice, it’s not ‘s’!)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Addr</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>char x [0]</td>
<td>100</td>
<td>‘t’</td>
</tr>
<tr>
<td>char x [1]</td>
<td>101</td>
<td>‘e’</td>
</tr>
<tr>
<td>char x [2]</td>
<td>102</td>
<td>‘s’</td>
</tr>
<tr>
<td>char x [3]</td>
<td>103</td>
<td>‘t’</td>
</tr>
<tr>
<td>char x [4]</td>
<td>104</td>
<td>‘\0’</td>
</tr>
</tbody>
</table>

Q: What’s the difference between “char x[5]” and a declaration like “char *x”?

For loop that iterates from 0 to COUNT-1.
At this point we have seen a few basic types, arrays, pointer types, and structures. So far we’ve glossed over how types are named.

```c
int x;        /* int; */
int *x;       /* pointer to int; */
int x[10];    /* array of ints; */
int *x[10];   /* array of pointers to int; */
int (*x)[10]; /* pointer to array of ints; */
```

C type names are parsed by starting at the type name and working outwards according to the rules of precedence:

- `int x[];` is an array of `int`
- `int (*x)[10];` is a pointer to an array of `int`
- `int (*x)[10];` is a pointer to an array of `int`

Arrays are the primary source of confusion. When in doubt, use extra parens to clarify the expression.

REMEMBER THIS: `()`, which stands for function, and `[]`, which stands for array, have higher precedence than `*`, which stands for pointer.

typedef defines a new type

typedef defines a new type
Another less obvious construct is the “pointer to function” type. For example, qsort: (a sort function in the standard library)

```c
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
/* function matching this type: */
int cmp_function(const void *x, const void *y);
/* typedef defining this type: */
typedef int (*cmp_type) (const void *, const void *");
/* rewrite qsort prototype using our typedef */
void qsort(void *base, size_t nmemb, size_t size, cmp_type compar);
```

- `void *` is a pointer to memory of unknown type.
- `size_t` is an unsigned int.
- `const` means the function is not allowed to modify memory via this pointer.
- The last argument is a comparison function.
References

• Linux/Unix Introduction
  – http://www.ee.surrey.ac.uk/Teaching/Unix/

• VI Editor
  – https://www.cs.colostate.edu/helpdocs/vi.html

• C Programming Tutorial

• Compiler, Assembler, Linker and Loader: A Brief Story
  – http://www.tenouk.com/ModuleW.html