Refresher Module CS231

Introduction to C

INDRAPRASTHA INSTITUTE *of*
INFORMATION TECHNOLOGY **DELHI**

Dynamic Memory Allocation

1. Malloc
2. Calloc
3. Realloc
4. Free

Dynamic Memory Allocation: Malloc

The word “malloc” stands for memory allocation. We use malloc to dynamically allocate a single large block of memory with the specified size.

- malloc returns a **pointer of type void** which can be **cast into a pointer of any form**.
- malloc **doesn’t initialize memory at execution time**.
- If space is insufficient, allocation fails and returns a NULL pointer.

Dynamic Memory Allocation: Malloc

The word “malloc” stands for memory allocation. We use malloc to dynamically allocate a single large block of memory with the specified size.

- malloc returns a **pointer of type void** which can be **cast into a pointer of any form**.
- malloc **doesn’t initialize memory at execution time**.
- If space is insufficient, allocation fails and returns a NULL pointer.

```
int *ptr = (int *) malloc (16*sizeof(char));
```

Here ptr is typecast into an integer pointer. We have reserved **16*1** bytes worth of uninitialized memory.

Dynamic Memory Allocation: Calloc

The word “calloc” stands for **contiguous** memory allocation. We use malloc to dynamically allocate a single large block of memory with the specified size.

- calloc returns a **pointer of type void** which can be **cast into a pointer of any form**.
- calloc initializes each block with a default value ‘0’.
- If space is insufficient, allocation fails and returns a NULL pointer.

```
int *ptr = (int *) calloc (16, sizeof(int));
```

Here ptr is typecast into an integer pointer. We have reserved **`16 contiguous segments in memory each with 4`** bytes worth of memory initialized to 0.

Dynamic Memory Allocation: Realloc

The word “realloc” stands for re-allocation of previously allocated memory. We use realloc primarily when the original malloc or calloc was too small.

- realloc returns a **pointer of type void** which can be **cast into a pointer of any form**.
- realloc does not initialize each block.
- If space is insufficient, allocation fails and returns a NULL pointer.

```
int *ptr = (int *) malloc (16*sizeof(int));

ptr = realloc (ptr, 32*sizeof(int));
```

Here ptr is typecast into an integer pointer. We have reserved **16*4** bytes worth of memory initially. We are then reallocating **32*4** bytes worth of memory and pointing it to ptr.

Dynamic Memory Allocation: Free

The free command is used to de-allocation of previously allocated memory.

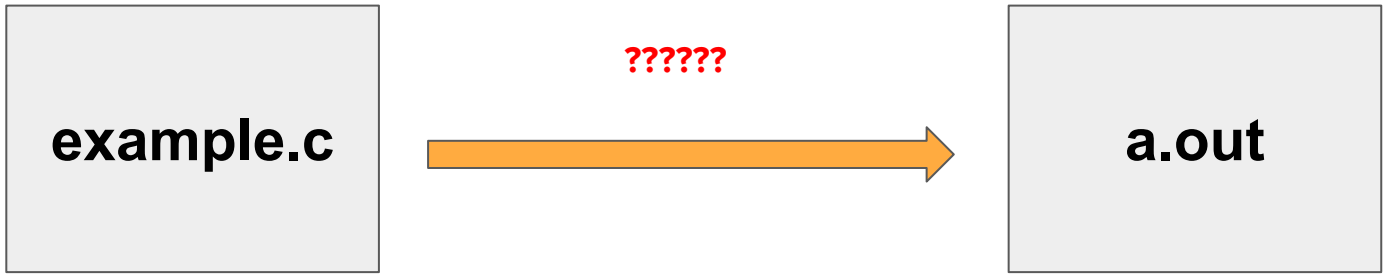
```
int *ptr = (int *) malloc (16*sizeof(int));

free(ptr);
```

Here ptr is typecast into an integer pointer. We have reserved **16*4** bytes worth of memory initially. We are then **freeing** the memory.

Stages of compilation

How do you go from a .c file to the a.out executable?



Stages of compilation

1. **Preprocessing**
2. **Compilation (Core-Compilation)**
3. **Assembly**
4. **Linking**

Stages of compilation

1. **Preprocessing**
   - input: .c file
   - output: **.i file**
2. **Compilation**
   - input: .i file
   - output: **.s file** or **.o file**
3. **Assembly**
   - input: .s file
   - output: **.o file**
4. **Linking**
   - **input:** .o file
   - **output: a.out file**

Stages of compilation

1. **Preprocessing**
   - input: .c file
   - output: **.i file**
2. **Compilation**
   - input: .i file
   - output: **.s file** or **.o file**
3. **Assembly**
   - input: .s file
   - output: **.o file**
4. **Linking**
   - **input:** .o file
   - **output: a.out file**

5. Run-Time Linking (loads and links shared libraries used by a.out)

Stages of compilation (use --save-temps flag)

1. **Preprocessing (-E flag)**
   - input: **.c file**
   - output: **.i file**
2. **Compilation (-S flag;** view file using **objdump)**
   - input: **.i file**
   - output: **.s file**
3. **Assembly (-c flag;** view file using **hexdump)**
   - input: **.s file**
   - output: **.o file**
4. **Linking (**view file using **hexdump)**
   - **input: .o file**
   - **output: .out file**

Stages of compilation (use --save-temps flag)

1. **Preprocessing (-E flag)**
   - Preprocessor directives # are interpreted and **commands** are stripped out.
2. **Compilation (-S flag)**
   - Preprocessed code is translated to assembly instructions *specific to the target processor architecture.*
3. **Assembly (-c flag; view file using hexdump)**
   - Translate the assembly instructions to object code (actual instructions).
4. **Linking**
   - The object code generated has missing pieces of instructions. These are filled in during linking.

Preprocessing (use -E flag)

The file with the result of preprocessing usually has "**.i**" suffix
The preprocessor carries out the following operations:

1. Inclusion of header files ('**`#include`**')
2. Macros substitution ('**`#define`**')
3. Conditional compilation ('**`#if', '#ifdef', '#else', '#elif', '#endif`**').
4. The result of preprocessing is called a **translation unit**

Compilation (use -S flag)

The file with the result of compilation usually has "**.s**" suffix

- **.s** files are source code files written in assembly code.
- The files contain assembly instructions to the processor in sequential order and are typically compiled based on a selected architecture.
- Machine dependent instructions (I/O) and early initialization such as setting up cache and memory.

```
8d 4c 24 04             lea    0x4(%esp),%ecx
83 e4 f0                and    $0xfffffff0,%esp
ff 71 fc                pushl  -0x4(%ecx)
55                      push   %ebp
89 e5                   mov    %esp,%ebp
51                      push   %ecx
83 ec 34                sub    $0x34,%esp
65 a1 14 00 00 00       mov    %gs:0x14,%eax
89 45 f8                mov    %eax,-0x8(%ebp)
31 c0                   xor    %eax,%eax
```

Assembly (use -c flag)

The file with the result of assembly usually has "**.o**" suffix

- **.o** files are relocatable object codes.
  - Relocatable code is software whose execution address can be changed. A relocatable program might run at address 0 in one instance, and at 10000 in another.

- Translation of the .**s** file to its corresponding binary machine code equivalent

Linking (use -o flag)

The link editor creates an executable file (**a.out** file) from one or more .o files

```
gcc -o simple simple.o
```

creates an executable file from `simple.o` and some standard libraries that gcc automatically links in

Linking (use -o flag)

The link editor creates an executable file (**a.out** file) from one or more .o files

```
gcc -o simple simple.o
```

creates an executable file from `simple.o` and some standard libraries that gcc automatically links in

Further Reading:

1. The runtime linker and dynamically linked libraries (**.so** files)
2. Extracting Information from **.o** and **a.out** files using **`hexdump, strings, objdump`**, etc. (https://www.cs.swarthmore.edu/~newhall/unixhelp/binaryfiles.html)
3. Use **`nm`** (or **`objdump -t`**) to list the symbol table from an **a.out** or **.so** file.
4. The **a.out** file is in the ELF file format which you can partially read using **`readelf`** and **`objdump`**.

Makefiles

1. What are makefiles?

2. Why makefiles?

3. How to create makefiles?

Makefiles

1. **What are makefiles?**
    - A makefile is a special file, containing shell commands named makefile.
    - Type make and the commands in the makefile will be executed.
2. **Why makefiles?**

3. **How to create makefiles?**

Makefiles

1. **What are makefiles?**
   - A makefile is a special file, containing shell commands named makefile.
   - Type make and the commands in the makefile will be executed.
2. **Why makefiles?**
   - Typing gcc for each individual file is not always a good idea;

     ex - files with dependencies, order of compilation matters; many files in project; etc.
3. **How to create makefiles?**

Makefiles

1. **What are makefiles?**
   - A makefile is a special file, containing shell commands named makefile.
   - Type make and the commands in the makefile will be executed.
2. **Why makefiles?**
   - Typing gcc for each individual file is not always a good idea;

     ex - files with dependencies, order of compilation matters; many files in project; etc.
3. **How to create makefiles?**
   - The makefile contains a list of rules. These rules tell the system what commands you want to be executed.
   - ```
     DEPENDENCY LINE   TARGET FILES:SOURCE FILES
     ACTION LINE       [tab]ACTION LINE(S)
     ```
     ```

Makefiles

Further Reading:

1. Makefile Tutorial
2. What is a Makefile?.

1. What are makefiles?

- A makefile is a special file, containing shell commands named makefile.
- Type make and the commands in the makefile will be executed.

2. Why makefiles?

- Typing gcc for each individual file is not always a good idea;

  ex - files with dependencies, order of compilation matters; many files in project; etc.

3. How to create makefiles?

- The makefile contains a list of rules. These rules tell the system what commands you want to be executed.
- ```
  DEPENDENCY LINE   TARGET FILES:SOURCE FILES
  ACTION LINE       [tab]ACTION LINE(S)
  ```
  ```

GDB

- GDB stands for the **GNU debugger**.
  - It supports Assembly, C, C++, Fortran, Go, Objective-C, Pascal, Rust and *others*.
  - GDB also has an inbuilt python interpreter.

- Compile using the **-g flag**.
  - This stops the compiler from optimizing and preserves debugging information.
- Run gdb using
  - gdb a.out
  - gdb a.out --tui (my recommendation)

GDB (basic commands)

- run r
- break <line number>/<function name> b
- next n
- list (not necessary if we start in tui mode)
- quit q
- print p
- info locals / breakpoints / frame

GDB (debugging commands)

- continue
- up/down
- backtrace bt
- step
- set <variable id>
- watch <variable id>
- what <variable>
- display <variable>/ undisplay <variable id>

Valgrind

What is valgrind?

It is a tool for checking memory management and threading bugs.

How to use valgrind?

valgrind ./a.out

Valgrind

What is valgrind?

It is a tool for checking memory management and threading bugs.

How to use valgrind?

valgrind ./a.out

Further Reading: Understanding Valgrind Output.

Recap

Thank You!