UNIX and C Programming (COMP1000)

Lecture 3: Pointers

Updated: 13th August, 2019

Department of Computing
Curtin University

Textbook Reading (Hanly and Koffman)

For more information, see the weekly reading list on Blackboard.

- **Chapter 6: Pointers and Modular Programming**
  Section 6.1 *also* introduces "Pointers to Files", which you can safely ignore until lecture 5.
- **Appendix A: More About Pointers**
  This is important material, even though it's in an appendix.

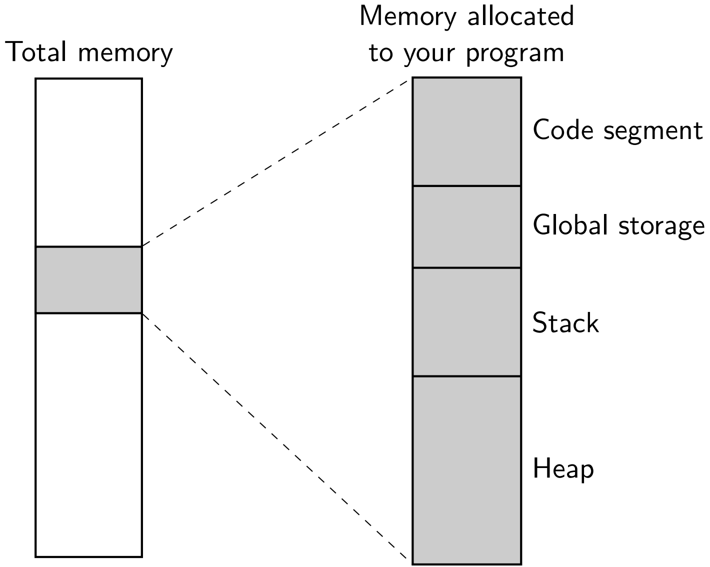
For Test 1, revise everything up to, and including, next week's lecture.

Outline

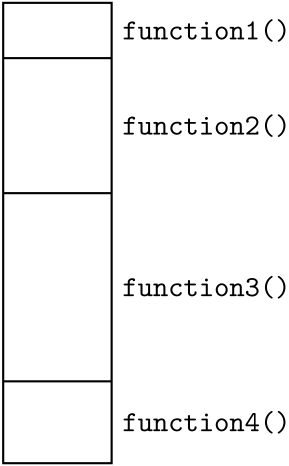
Memory

- Essentially a single giant sequence of bytes.
- Each byte has a unique, sequential "address".
- Carved up in complex ways, into many parts.

Memory Structure



Code Segment

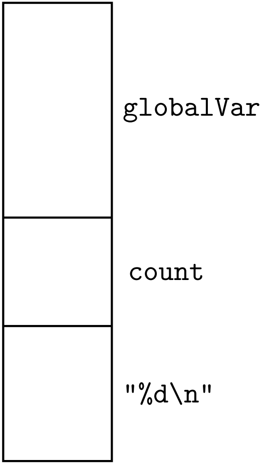


- Read-only.
- Consists of a sequence of functions.
- Each function is a sequence of machine code instructions.
- Each instruction occupies one or more bytes.
- The last instruction is the "return" instruction.

Global Storage

- Contains statically-allocated data, including:
  - Global variables.
  - Local static variables.
  - String literals.
- Writable (not read-only), but has a fixed size

Global Storage — Example

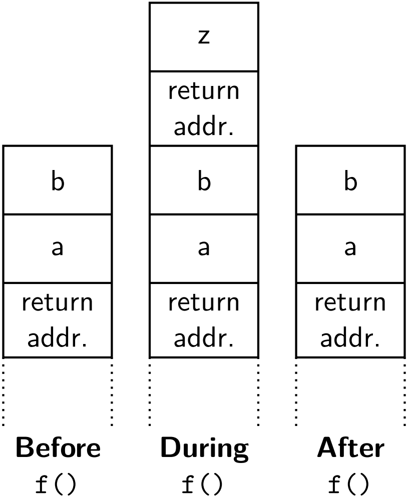



```c
#include <stdio.h>

double globalVar = 1.5;

void printNext(void)
{
    static int count = 0;
    count++;
    printf("%d\n", count);
}
```

The Stack

- "Last in, first out"
- Used to store:
  - Temporary/intermediate calculation results
  - Local variables (except static variables)
  - Function parameters (sometimes)
  - Function return location (i.e. where to jump back to when a function finishes)
- Grows and shrinks over time
- Grows when a function is called (when local variables are allocated)
- Shrinks when a function returns (when local variables are destroyed)

The Stack — Example




```c
int a = 10;
int b = 12;
...
f(a, b);
...

void f(int x, int y)
{
    int z = x / y;
    printf("%d\n", z);
}
```

The Heap

- A giant pool of dynamically-allocated memory.
- No particular fixed size or structure.
- Allocated and freed as needed by the program.
- Discussed later in this lecture.

The Registers

- Not part of main memory; found inside the CPU itself.
- Extremely small amount of storage — only a few bytes.
- Very fast.
- Used to store (depending on the compiler and if there's enough room):
  - Function parameters.
  - Temporary/intermediate calculation results.
  - Variables declared with the `register` storage class.
- Registers also keep track of:
  - The location of the stack.
  - The currently-executing instruction.
- For most programming purposes, we can usually ignore the registers.

Pointers

- Each byte in memory has a unique, sequential "address".
- These addresses themselves are often stored in memory.
- This is called a "pointer".
- Where a value occupies multiple bytes (as most do), a pointer will point to the first byte (i.e. lowest memory address).

Memory (in 4-byte chunks or "words")

0x1000		
0x1004		
0x1008	35.8	ordinary `float` variable
0x100C		
0x1010		
0x1014	0x1008	pointer (to above variable)
0x1018		

Pointer Data Types

- Pointers are a data type, or rather a *set of data types*.
- To declare a pointer, we must also declare what sort of data it points to.

```
int* size;
double* speed;
char* letter;
```

- A pointer to an `int` and a pointer to a `float` are different data types.

Pointer Declaration Syntax

data-type * name;

- Declares a variable ("**name**"), storing a memory address.
- That memory address itself stores a value of type "**data-type**" (e.g. `int`, `float`, etc.).
- Spacing is meaningless. These are all equivalent:

```
int*x;
int *x;  /* Widely used. */
int* x;  /* Used by me (to avoid confusion). */
int * x;
```

- Later we'll see "`*x`" used like a variable, but be careful:
  - The real variable is still `x`, not `*x`.
  - The `*` has *another* meaning: "dereference".

Multiple Declarations

- You can declare multiple pointers on the same line:

```
float *ptr1, *ptr2, *ptr3;
```

- Notice that they each require a `*`.
- The `*` is part of the data type, but attaches to one name only.
- Consider this:

```
float* ptr, number;
```

`ptr` is a pointer, but `number` is just a `float`.

NULL

- A special pointer value — a "pointer to nothing".
- Useful for initialising pointers.
- Actually a preprocessor macro, defined in all the standard header files.
- Like Java's "`null`". *Not* like SQL's "NULL".

Extra Note

- `NULL` is *often* defined to be zero (on many platforms).
- Recall that `FALSE` is also zero (from the previous lecture).
- This fact is often used (or misused) inside `if`, `while` and `do-while` statements.

Dereferencing

The Uses of *

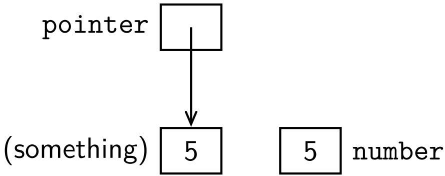
1. Multiplication (obviously);
2. Declaring a pointer (e.g. `int* ptr`); and
3. "Dereferencing" (following) a pointer.

The Dereference Operator

- After you've created a pointer, you can use it to access the memory location it points to ("dereferencing").
- Place `*` in front of a pointer variable; e.g. "`*ptr`".
- "`*ptr`" is what `ptr` points to. If `ptr` points to an `int`, then `*ptr` *is* an `int`.
- You can use `*ptr` just like an ordinary variable.
  - But remember that `ptr` is the real variable.

Dereferencing Example

```
int* pointer;
int number;

...
/* Assume 'pointer'
points to a valid
memory address! */

*pointer = 5;
number = *pointer;
```



- 5 is stored at the memory address contained in `pointer`.
- `number` is assigned the same value.

Declaration or Dereference?

- You can declare and initialise a pointer on one line:

```
int* x = y;
```

OR

```
int *x = y; /* Remember: spacing is meaningless.*/
```

- Be careful – this does *not* dereference `x`!
- It *creates* `x`, with the datatype `int*` (a pointer to an integer).
- It also initialises `x` with the value of `y`.
  - `y` must be another pointer of the same type (`int*`).
- The following is equivalent to the above:

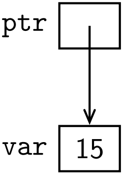
```
int* x;
x = y;
```

Obtaining Pointers — The Address-Of Operator

- How do you make a pointer point to something?
- Use the & (address-of) operator to get the location of a variable.
- Place a "&" in front of a variable name; e.g. "&var".
- "&var" is a pointer to var.

Example

```
int var = 15;
int* ptr = &var;
```



Now, *ptr *is* var!

Referencing and Dereferencing

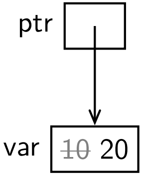
- The `*` and `&` operators are opposites.
- `&` gets an address, while `*` follows an address to get a value.
- `*&var` is equivalent to just `var`.
- `&*pointer` is equivalent to `pointer`.

Order of Operators

- Consider this: `*ptr++;`
- Dereference 1st and increment 2nd?
- Or increment 1st and dereference 2nd? (Correct!)
- To avoid ambiguity/confusion:
  - Use brackets: `(*ptr)++;`
  - Or use pre-increment: `++*ptr;`

Pointer Example

```
int var = 10;
int* ptr;

ptr = &var; /* Make ptr point to var */
*ptr = 20;  /* Set var to 20 */

printf("%d\n", var); /* Prints "20". */
```



- The line "`ptr = &var`" stores the address of `var` in `ptr`.
- Thus, `*ptr` *is* `var`.
- Thus, "`*ptr = 20`" actually modifies `var`.

Another Pointer Example

```c
int b = 15;
int c = 30;
int* x;
int* y;

x = &b;    /* Make x point to b */
y = x;     /* Make y point to whatever x points to */
x = &c;    /* Make x point to c */

printf("%d\n", *x);      /* Prints "30". */
printf("%d\n", *y);      /* Prints "15". */
```

At the end:

- x points to c
- y points to b

L-values

- Consider the humble assignment statement:

```
x = y + 5;   /* Changes x to be equal to y + 5. */
```

- The right side can be a complex expression, whereas the left side is often just a variable name...
- ...but not always!
- Expressions are allowed on the left, *if* they are "L-values".
- An L-value is a value *plus* a memory location that holds it.
  - e.g. `x` and `y`. Variables have values plus memory locations.
- Non-L-values include:
  - 5 – literal numbers have no memory location.
  - `y + 5` – arithmetic has no memory location.
  - (The compiler could choose to store these in memory, but only temporarily, and it may not need to at all.)

L-values and Pointers

- The dereference (*) operator creates an L-value.

```
int num = 42;
int* ptr = &num;
...
*ptr = 64;
```

  - "*ptr" is an expression, not a variable.
  - But it's an L-value, so it can appear on the left.
  - *ptr has a value (42) *and* a memory address (that of num).
- The address-of (&) operator *requires* an L-value.

```
int num = 42;
int* ptr;
...
ptr = &num;
```

  - You cannot write &42 or &(num + 5). They have no meaning.

Passing by Value

- Function parameters are usually "passed by value".
- A function's parameters are *copies* of the values passed into it.
- The function can overwrite the copies without affecting the originals.

Example

```
double invert(double x) {
    x = 1.0 / x;
    return x;
}
...
double a = 0.5, b;
b = invert(a);
```

Afterwards, `a == 0.5` and `b == 2.0`.

Passing by Reference

- Using pointers, parameters can be “passed by reference”.
- Instead of supplying a copy of the value, you supply a pointer to the original.
- The function can then change the original variable.

Example

```
void invert(double* x) {
    *x = 1.0 / *x;
}
...
double a = 0.5;
invert(&a);
```

The value of “a” changes from 0.5 to 2.0.

Passing by Reference (2)

- Used when the value to be passed in occupies many bytes (avoids unnecessary copying).
- Used when you want the function to export *more than one* value.
- In C, arrays and strings can only be passed by reference (but we'll get to that later).

Example — `scanf()`

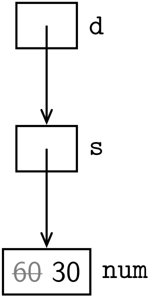
- Recall that `scanf()` requires a `&` in front of its parameters.
- These parameters are exports, not imports.
- `scanf()` needs memory addresses to store values.

Pointers to Pointers

- A pointer can point to any data type, including another pointer.
- This is called a "pointer to a pointer (to another datatype)".
- (a "pointer to a pointer to a pointer...", etc. are also possible.)
- We will learn more about their uses later in the unit.

Example

```
int num = 60;
int* s;      /* Pointer to an integer */
int** d;     /* Pointer to a pointer */

d = &s;      /* Make d point to s */
*d = &num;   /* Make s point to num */
**d = 30;    /* Assign 30 to num */
```



Void Pointers

- A pointer to "`void`"; a generic pointer.
- Contains a valid memory address, but the type of data stored there is unspecified.
- Cannot be dereferenced (directly):

```
void assign(int* intPtr, void* voidPtr)
{
    *intPtr = 42;    /* Perfectly fine. */
    *voidPtr = 42;  /* Illegal. */
}
```

  - We can't write `*voidPtr`, because we don't know the datatype.
- To use `void*`, it must be *typecast* to something else.

Void Pointers and Typecasting

- Pointers can be typecast to different kinds of pointers:

```
void assign(void* voidPtr) {
    int* intPtr;
    intPtr = (int*)voidPtr; /* Typecast & assign*/
    *intPtr = 42;
}
```

  - Here, we copy `voidPtr` to `intPtr`.
  - The typecast is highlighted.
  - Then we can access the memory.
  - We *assume* `voidPtr` really points to an int (exercise caution).
- We don't even need `intPtr`, actually:

```
void assign(void* voidPtr) {
    *(int*)voidPtr = 42; /* Same as above. */
}
```

Void Pointers — Example

- Void pointers are sometimes used to handle multiple datatypes:

```
void printN(char format, void* n) {
    if(format == 'i') {
        /* Assume n really points to an int. */
        printf("%d\n", *(int*)n);
    }
    else if(format == 'f') {
        /* Assume n really points to a float. */
        printf("%f\n", *(float*)n);
    }
}
```

- Here, `printN()` can be used to print ints and floats.
- The `format` parameter tells us how to treat the void pointer.

Invalid Pointers

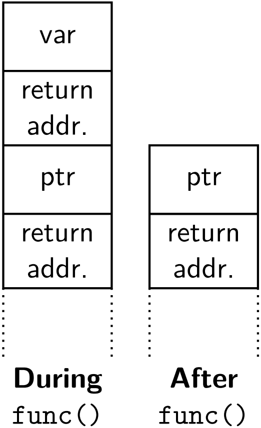
- C offers no guarantee that a pointer points to anything meaningful.
- It's your job to make sure it does.
- Newer languages (like Java) use pointers "behind the scenes", and so protect you from misusing them.
- C offers no protection.

Example

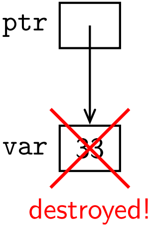
Say your function returns a pointer to one of its own local variables:

- The local variable will be destroyed when the function ends.
- The pointer will point to memory no longer in use.

Invalid Pointers — Example




```
int* ptr;
ptr = func();
...

int* func(void)
{
    int var = 33;
    return &var;
}
```



Heap Memory

- So far we've dealt with the stack and global storage.
- The "heap" is a much larger (and more flexible) pool of memory.
- Your functions can return pointers to heap-allocated memory.
  - You don't have to allocate and use the memory in the same function.
  - Can't be done with stack-based variables.
- Mostly useful for arrays (lecture 4) and structs (lecture 6).

Allocation

- All memory must be "allocated" before it can be used.
- Stack memory is allocated simply by declaring a local variable.
- Heap memory is allocated by calling the `malloc()` function.
- The OS finds a block of memory of a suitable size, and grants your program permission to access it.
- This memory *stays* allocated until you "free" it — it doesn't disappear when a function ends.

Deallocation / Freeing

- Newer languages (like Java) have "garbage collection".
  - When a block of heap memory no longer has any pointers to it, it is automatically freed.
- C **does not** have garbage collection!
- You must explicitly deallocate heap memory when you're done with it.

The Operating System

- When a program ends, all its allocated memory *is* forceably freed by the OS.
- For large or long-running programs, this doesn't help you.
- Don't rely on it!

Memory Leaks

- If you forget to deallocate heap memory in C, you get a "memory leak".
- This is often an invisible error.
- Your program may appear to work perfectly.
- However, it may consume much more memory than needed (and you may run out of memory).
- There are tools — like `valgrind` — to help detect memory leaks.

Heap Allocation in C

- Heap allocation (and de-allocation) is done through specific functions.
- Mainly uses the `stdlib.h` library.
- Some generic memory-related functions are found in `string.h`.
  - (These include `memset()` and `memcpy()`, which we'll cover next week.)

The `sizeof` Operator

- Reports the size (in bytes) of any data type.
- A C language construct (*not* a function).
- On a typical 32-bit machine:

```
sizeof(char)     == 1
sizeof(short)    == 2
sizeof(int)      == 4
sizeof(float)    == 4
sizeof(double)   == 8
sizeof(void*)    == 4
sizeof(short**)  == 4
```

- These sizes may change across different hardware — that's why `sizeof` is useful!
- A special unsigned integer type "`size_t`" represents memory sizes:

```
size_t numBytes = sizeof(long);
```

Heap Allocation — `malloc()`

- The `malloc()` function allocates a block of heap memory.
- Takes an `int` parameter — the size in bytes.
- Returns a `void*` pointing to the newly-allocated memory.
- Returns `NULL` if the memory could not be allocated.

Usage

Since `malloc()` doesn't know (or care) what you want to store in the memory:

- Use `sizeof` to determine the number of bytes to allocate.
- Typecast the returned `void` pointer to the appropriate pointer type.

malloc() Example

To *dynamically* allocate storage for various values:

```
#include <stdlib.h>
...
int* integer = (int*)malloc(sizeof(int));
float* real = (float*)malloc(sizeof(float));
double* bigReal = (double*)malloc(sizeof(double));
```

- "`sizeof(int)`" — the number of bytes to allocate.
- "`(int*)` — typecast to an `int` pointer.
- Once allocated, you can use `*integer`, `*real` and `*bigReal` just like "ordinary" variables.

Deallocation — `free()`

- Every block of memory allocated with `malloc()` must eventually be "freed".
- The `free()` function takes a pointer to the block, and frees it.
- Returns nothing.
- Thereafter, the pointer is invalid.
- Don't free memory before you're finished with it!

`NULL`

- It's good practice to set a freed pointer to `NULL`.
- You should do this immediately after a call to `free()`.
  - (Unless the pointer variable itself is about to disappear too.)

free() — Example

```c
#include <stdlib.h>

int main(void) {
    double* real = (double*)malloc(sizeof(double));

    ... /* Use the memory */

    free(real);
    real = NULL;

    ... /* Do something else afterwards */
}
```

- You don't need to tell `free()` how big the block is.
- Make sure you set the pointer to `NULL` afterwards.

Valgrind

- It's easy to make mistakes with memory in C:
  - Using uninitialised values,
  - Accessing unallocated memory,
  - Failing to free memory,
  - Freeing the same memory more than once,
  - Losing track of allocated memory (memory leaks).
- The `valgrind` tool helps you find these.
- `valgrind` works with compiled programs.
- It detects memory errors while a program is running.
- To use it, type:

```
[user@pc]$ valgrind [options] ./program ...
```

Leave **[options]** blank for a summary of the errors.

Debugging Information

- To get the most out of `valgrind`, you need "debugging information".
- This is a compile option, which inserts extra information into the executable file.
- On the command-line:

```
[user@pc]$ gcc -g -c file.c
```

- In a Makefile:

```
CFLAGS = -Wall -pedantic -ansi -g
```

- Without this, `valgrind` can't give you any line numbers.

Valgrind Output

This is `valgrind` detecting a memory leak:

```
LEAK SUMMARY:
   definitely lost: 10 bytes in 1 blocks
   indirectly lost: 0 bytes in 0 blocks
     possibly lost: 0 bytes in 0 blocks
   still reachable: 0 bytes in 0 blocks
        suppressed: 0 bytes in 0 blocks
Rerun with --leak-check=full to see details of
leaked memory
```

Then, to find *where* the leak occurred:

```
[user@pc]$ valgrind --leak-check=full ./program ...
```

Coming Up

- Next week's lecture will expand on pointers, covering arrays and strings.
- Once again, make sure you complete the tutorial exercises.