last update: 18th February 2024

Session: 2023-24

COMP281 Lecture 7

Principles of C and Memory Management

Phil Jimmieson

UNIVERSITY OF LIVERPOOL

Department of Computer Science

Last Lecture

- Arrays
- Debugging

Last Lecture

- Arrays
- Debugging

Today

- Pointer Basics

Pointer Basics

What is a Pointer?

What is a pointer in C?

- A variable holding the **address** of another variable of same data type
  - To access **memory** and manipulate the address
- One of the most distinct and *exciting* features of C language they
  - Provide power and flexibility to the language
  - But, with increased power comes increased responsibility
    - Allow new and more ugly types of bugs
    - Pointer bugs can crash in random ways which make them very difficult to debug

Why have pointers?

- Pointers solve 2 common software problems
  - Allow different sections of code to share information easily (you can get the same effect by copying information back and forth, but pointers solve the problem better)
  - Enable complex “linked” data structure, such as
    - Linked lists
    - Binary trees

Why have pointers?

- Pointers solve a hardware problem too
  - Allow interaction with other hardware that has access to the system memory - DMA (Direct Memory Access)
  - e.g. I/O that can be done independent of the CPU until the data is transferred.
  - The CPU can put the data in a location, using a pointer, alert the DMA device and then do some work while the other device is handling the data.
  - When all the data has been transferred, you can use the pointer to access it.

Address

- Whenever a variable is defined, a **memory** location is assigned for it, where its value will be **stored**.
- You can easily check this memory address using the & symbol.

Example 1

```
#include <stdio.h>
int main() {
    int x = 9;
    printf("Value of the variable x is: %d\n", x);
    printf("Memory address of the variable x is: %p\n", &x);
    return 0;
}
```

Note:
%p format specifier
for addresses

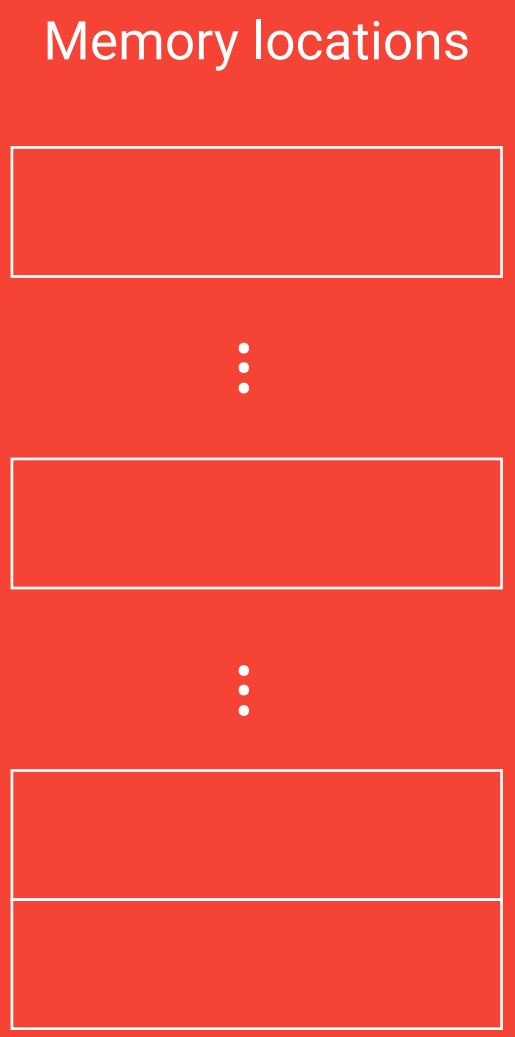
Output

```
Value of the variable x is: 9
Memory address of the variable x is: 0x7ffee3aec8d8
```

Address vs pointer

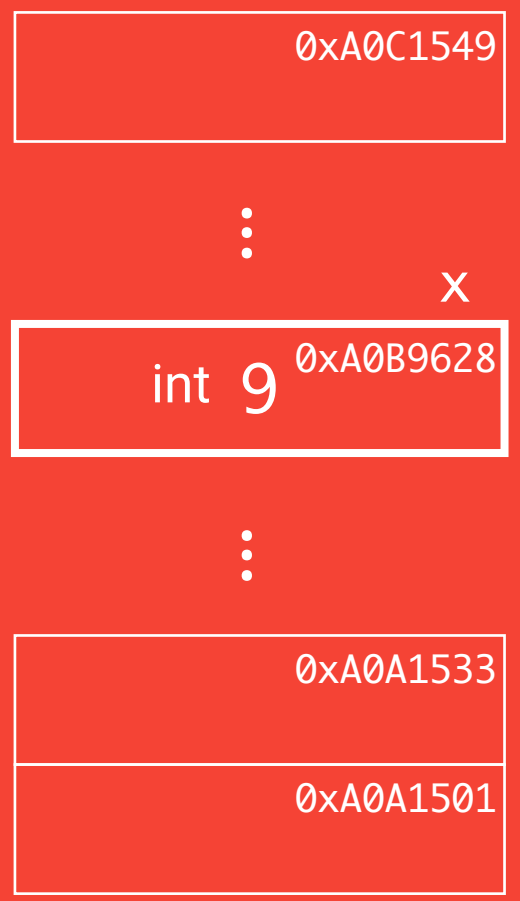
- Address
  - A location in memory where data can be stored
  - e.g. of a variable or an array
  - Address of variable x is written as &x
- Pointer
  - A variable which holds an **address**

When a variable is defined in a program, the machine allocates a location i.e. an address for that variable in memory, to hold the assigned value. This location has its own address number.



When a variable is defined in a program, the machine allocates a location i.e. an address for that variable in memory, to hold the assigned value. This location has its own address number.

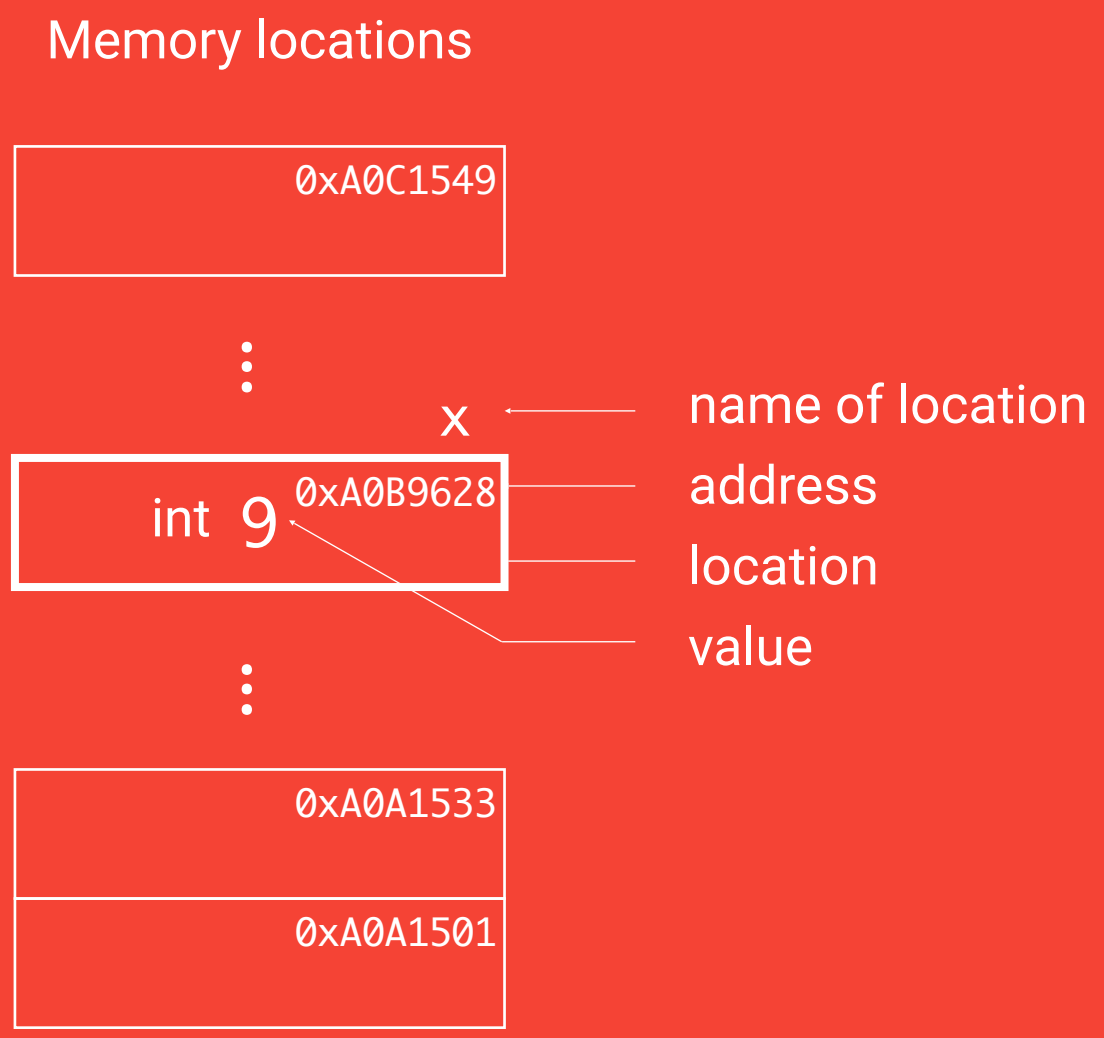
When a variable is defined in a program, the machine allocates a location i.e. an address for that variable in memory, to hold the assigned value. This location has its own address number.




```
int x = 9;
```

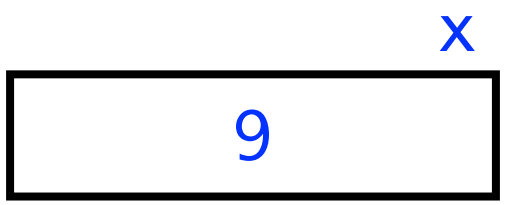
(define an `int` type variable)

When a variable is defined in a program, the machine allocates a location i.e. an address for that variable in memory, to hold the assigned value. This location has its own address number.



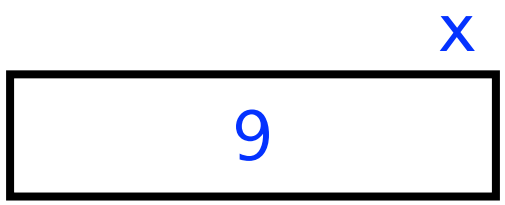
Variable vs pointer

- Simple `int` (and other) **variables** operate pretty intuitively. An `int` variable is like a box which can store a single `int` value such as 9.



Variable vs pointer

- Simple `int` (and other) **variables** operate pretty intuitively. An `int` variable is like a box which can store a single `int` value such as 9.



- A **Pointer** works a little differently.
  - It does not store a simple value directly
  - Instead it stores a reference to another value
  - The variable the pointer refers to is known as its "pointee"

Variable vs pointer variable

Name	Address	Content	
`int x = 9;`	0xA0B9628	9	x
`int *y = &x;`	0xA0C1549	0xA0B9628	y

```
/*  y "points" to x  */
```

Variable vs pointer variable

Name	Address	Content	
`int x = 9;`	`0xA0B9628`	`9`	`x`
`int *y = &x;`	`0xA0C1549`	`0xA0B9628`	`y`

```
/*  y "points" to x  */
```

`x` is the **name** of the variable

`&x` is the **address** of the variable `x`

`*y` is the **content of address** stored in pointer variable `y`

`*` operator **dereferences** the pointer `y`

The * operator

- The many meanings of the * operator
  - Multiplication

    ```
    a = b * c;
    ```

  - Declaring a pointer variable

    ```
    int *a;
    ```

  - Dereferencing a pointer

    ```
    printf("%d", *a);
    ```

The * operator

- The many meanings of the
  - Multiplication

    ```
    a = b * c;
    ```

  - Declaring a pointer variabl

    ```
    int *a;
    ```

  - Dereferencing a pointer

    ```
    printf("%d", *a);
    ```

"Dereferencing"?

The "dereference" operation follows a pointer's reference to get the value of its *pointee*.

```
int x = 9;
int *y = &x;
```

The value of the dereference of `y` is `9`. The "dereference" operation accesses the value of the *pointee*.

The only **restriction** is that the pointer must have a *pointee* for the dereference to access. Almost all bugs in pointer code involve violating that one restriction. A pointer **must** be assigned a *pointee* before dereference operations will work.

Declaration of pointer variable

- General syntax of pointer declaration

```
datatype *pointer_name;
```

- The data type of a pointer must be same as the data type of the variable to which the pointer variable is pointing.
- Some examples

```
int *ip;     /* pointer to integer variable */
float *fp;   /* pointer to float variable */
char *cp;    /* pointer to char variable */
```

Declaration of pointer variable

- Declaring multiple Pointer variables

```
int *a, *b;      /* a, b are pointers to int */
```

- If you do this

```
int *a, b;       /* b is just an int! */
```

- Then only the first variable will be a pointer
- Rule: every pointer variable in declaration must be preceded by a *

Initialisation of pointer variable

- Pointer initialisation is the process of assigning the address of a variable to a pointer variable.
- Pointer variable can only contain an address of a variable of the same data type.
- The address operator **&** is used to determine the address of a variable.
- The **&** returns the address of the variable associated with it.

Initialisation of pointer variable

Some examples

```
int a;
int *ptr;     /* pointer declaration */
ptr = &a;     /* pointer initialisation */

float a;
int *ptr;
ptr = &a;     /* ERROR, type mismatch */
```

Note that

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

really means

```
int *y;  /* declare y as a pointer to int */
y = &x;  /* assign x's address to y */
```

- Don't confuse this `*` with a dereference!

Example 2

```
#include <stdio.h>
int main() {
    int   x =  9;
    int  *y = &x;
    int **z = &y;

    printf("%p\t%d\n", &x, x);
    printf("%p\t%p\t%d\n", &y, y, *y);
    printf("%p\t%p\t%p\t%d\n", &z, z, *z, **z);

    return 0;
}
```

Example 2

```c
#include <stdio.h>
int main() {
    int   x =  9;
    int  *y = &x;
    int **z = &y;

    printf("%p\t%d\n", &x, x);
    printf("%p\t%p\t%d\n", &y, y, *y);
    printf("%p\t%p\t%p\t%d\n", &z, z, *z, **z);

    return 0;
}
```

Output

```
0x7ffeebee48c8 9
0x7ffeebee48c0 0x7ffeebee48c8 9
0x7ffeebee48b8 0x7ffeebee48c0 0x7ffeebee48c8 9
```

Example 2

```
int   x =  9;
int  *y = &x;
int **z = &y;
```

Name	Address	Content
x	0x7ffeebee48c8	9
y	0x7ffeebee48c0	0x7ffeebee48c8
z	0x7ffeebee48b8	0x7ffeebee48c0

Initialisation of pointer variable

- If you're not sure about which variable's address to assign to a pointer variable in a declaration, assign a NULL value to it.
- A pointer which is assigned a NULL value is called a NULL pointer

```
int *ptr = NULL;
```

Example 3

```
#include <stdio.h>
int main() {
   int a, *p;                /* declaring the variable and pointer */
   a = 10; p = &a;           /* initialising the pointer */
   printf("%d\n", *p);       /* prints the value of a */
   printf("%d\n", *&a);      /* also prints the value of a */
   printf("%p\n", &a);       /* prints the address of a */
   printf("%p\n", p);         /* also prints the address of a */
   printf("%p\n", &p);       /* prints the address of p */
   return 0;
}
```

Example 3

```
#include <stdio.h>
int main() {
    int a, *p;                    /*declaring the variable and pointer*/
    a = 10; p = &a;
    printf("%d\n", *p);
    printf("%d\n", *&a);
    printf("%p\n", &a);
    printf("%p\n", p);
    printf("%p\n", &p);
    return 0;
}
```

Output

```
10
10
0x7ffee90d38c8
0x7ffee90d38c8
0x7ffee90d38c0
```

Pointer to Pointer (Double Pointer)

- Pointers are used to store the address of other variables of the same datatype. But if you want **to store the address of a pointer variable**, then you again **need a pointer to store it**.
- Thus, when one pointer variable stores the address of another pointer variable, it is known as **Pointer to Pointer** variable or **Double Pointer**.

Pointer to Pointer (Double Pointer)

- General syntax of Pointer to Pointer (Double Pointer)

```
int **p1;
```

- Here, two *indirection operators* (*) are used to store and point to the address of a pointer variable `int *`
- To store the address of this (Double Pointer) variable p1, the syntax is:

```
int ***p2;
```

Example 4

```
int  a = 10;
int  *p1;    /* this will be used to store the address of variable a */
int  **p2;

p1 = &a;
p2 = &p1;

printf("Address of a = %p\n", &a);
printf("Address of p1 = %p\n", &p1);
printf("Address of p2 = %p\n\n", &p2);

printf("Value at the address stored by p2 = %p\n", *p2);
printf("Value at the address stored by p1 = %d\n\n", *p1);

printf("Value of **p2 = %d\n", **p2);

/* This is not allowed, it will give a compile time error-
   p2 = &a;
   printf("%p", p2); */
```

Example 4

```
int  a = 10;
int  *p1;    /* this will be used to store the address of variable a */
int  **p2;

p1 = &a;
p2 = &p1;

printf("
printf("
printf("

printf("
printf("

printf("

/* This is not allowed, it will give a compile time error-
   p2 = &a;
   printf("%p", p2); */
```

Output:

```
Address of a = 0x7ffee1a7f8c8
Address of p1 = 0x7ffee1a7f8c0
Address of p2 = 0x7ffee1a7f8b8

Value at the address stored by p2 = 0x7ffee1a7f8c8
Value at the address stored by p1 = 10

Value of **p2 = 10
```

Mnemonics: fetch/store

- When using the `*` (dereference) operator in an **expression**, you **fetch** the contents at that address

```
printf("y's contents are: %d\n", *y);
```

- When using the `*` (dereference) operator on the left-hand side of the `=` sign in an assignment statement, you **store** into that address

```
*y = 10; /* store 10 into address pointed to by y */
```

Pointers – call by reference

- Pointers can be used for a non-obvious trick
- **Recall**: in C, variables are copied before being sent to a function
  - Referred to as "**call-by-value**"
- Significance is that passing a variable to a function cannot change the variable's value
- What if you *want* to change the variable's value inside the function?

Pointers – call by reference

```
void incr(int z) {
    z++;
}
```

Pointers – call by reference

```
void incr(int z) {
    z++;
}

int x = 10;

/* want to increment x */
incr(x);

/* What is x now? */
```

Pointers – call by reference

```
void incr(int z) {
    z++;
}

int x = 10;

/* want to increment x */
incr(x);

/* What is x now? */
/* Still 10: incr() does nothing */
```

Pointers – call by reference

```c
void incr(int z) {
    z++;
}

int x = 10;

/* want to increment x */
incr(x);

/* What is x now? */
/* Still 10: incr() does nothing */
```

```c
void incr(int *z) {
    (*z)++;
}

int x = 10;

/* want to increment x */
incr(&x);

/* What is x now? */
```

Pointers – call by reference

```
void incr(int z) {
    z++;
}

int x = 10;

/* want to increment x */
incr(x);

/* What is x now? */
/* Still 10: incr() does nothing */
```

```
void incr(int *z) {
    (*z)++;
}

int x = 10;

/* want to increment x */
incr(&x);

/* What is x now? */
/* Yep, it's 11 now! */
```

Pointers – call by reference

```
int x = 10;
incr(&x); /* want to increment x */
```

- Now you should be able to work out why this works
- Where have we seen this before?

```
int i;
scanf("%d", &i);  /* read in i */
```

Pointers – call by reference

- Easy mistake to make

```
void inc(int *z) {
    *z++;            /* Won't work! */
    /* Parsed as: *(z++) */
}
```

- Need to use `(*z)++` here
- Precedence rule: use `( )` if any confusion may exist

Summary

Today

- Pointer basics
  - variables vs pointers vs address vs double pointers

Next

- Pointer to Array
- Pointer Arithmetic
- Pointer with Functions