Computer Programming 143 – Lecture 18
Pointers I

Electrical and Electronic Engineering Department
University of Stellenbosch

Prof Johan du Preez
Mr Callen Fisher
Dr Willem Jordaan
Dr Hannes Pretorius
Mr Willem Smit

Copyright & Disclaimer

Copyright

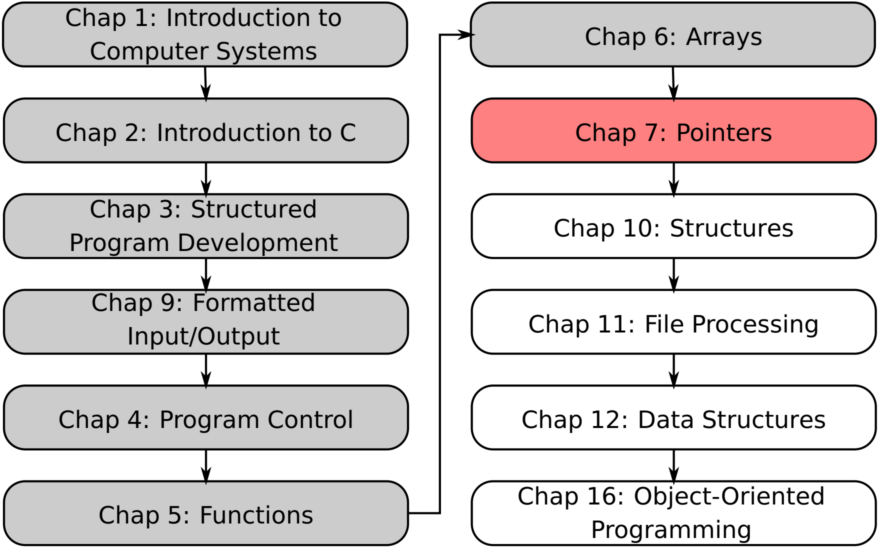
Disclaimer

This content is provided without warranty or representation of any kind. The use of the content is entirely at your own risk and Stellenbosch University (SU) will have no liability directly or indirectly as a result of this content.

The content must not be assumed to provide complete coverage of the particular study material. Content may be removed or changed without notice.

The video is of a recording with very limited post-recording editing. The video is intended for use only by SU students enrolled in the particular module.

Module Overview



Chap 1: Introduction to Computer Systems

Chap 2: Introduction to C

Chap 3: Structured Program Development

Chap 9: Formatted Input/Output

Chap 4: Program Control

Chap 5: Functions

Chap 6: Arrays

Chap 7: Pointers

Chap 10: Structures

Chap 11: File Processing

Chap 12: Data Structures

Chap 16: Object-Oriented Programming

Lecture Overview

1. Introduction (7.1)
2. Pointer Variable Declaration and Initialisation (7.2)
3. Pointer Operators (7.3)

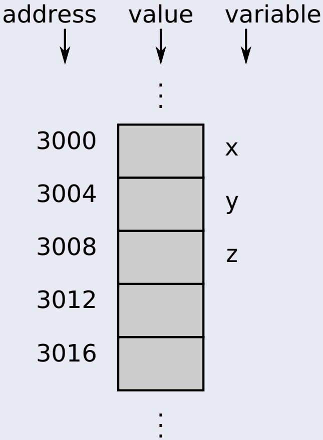
7.1 Introduction

Pointers

- So far, we have used two kinds of variables:
  - **scalar** variable: *single value*
  - **array** variable: *several values* (of same type)
- Today we encounter a whole new kind of variable:
  - **pointer** variable: *memory location of a value*
- Very powerful
- Simulate call-by-reference
- Close relationship with arrays and strings

7.2 Memory Concepts

Memory visualisation

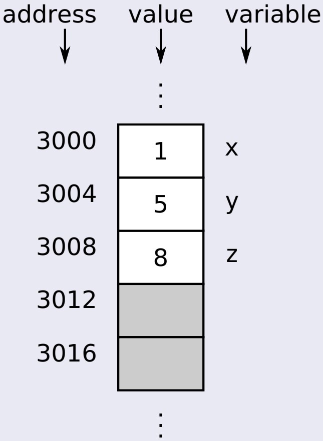


address	value	variable
⋮		
3000		x
3004		y
3008		z
3012		
3016		
⋮		

```c
int x, y, z;
```

7.2 Memory Concepts

Memory visualisation



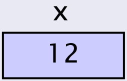
address	value	variable
	⋮	
3000	1	x
3004	5	y
3008	8	z
3012		
3016		
	⋮	

```c
int x, y, z;
x = 1;
y = 5;
z = 8;
```

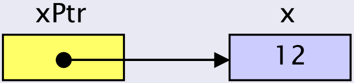
7.2 Pointer Variable Declaration and Initialisation I

Pointer variables

- Each variable has a name, value, and memory address
- Access variable's value through variable name (direct reference)
- (Before, values were numbers, characters, or arrays thereof)



- Pointers contain memory addresses as their values
- A pointer's value is the memory address of a(nother) variable
- That variable can then be accessed through the pointer (indirect reference)



- Indirection – Using an address (pointer) to access a variable

7.2 Pointer Variable Declaration and Initialisation II

Pointer declaration

- * used to declare pointer variables

```
int *myPtr;
```

Defines a pointer to an int value (pointer of type `int`)

- Multiple pointers require using a * before each variable definition

```
int *myPtr1, *myPtr2, myInt;
```

  - Pointer and scalar variables can be declared on the same line
- Can define pointers to any data type
- Initialise pointers to `0`, NULL, or an address
  - `0` or NULL – points to nothing (NULL preferred)

```
int *myPtr1 = NULL;
int *myPtr2;
myPtr2 = 0x22FF7C; /* Hardware address */
```

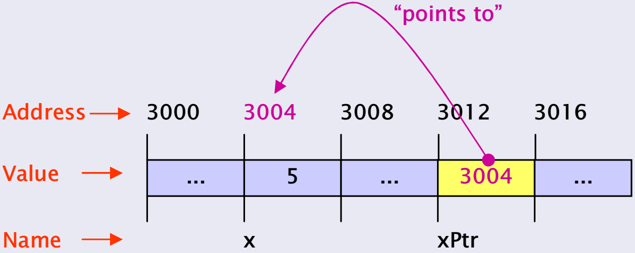
7.3 Pointer Operators I

& (address operator)

- Returns address of operand

```
int x = 5;
int *xPtr;
xPtr = &x;     /* value of xPtr becomes address of x */
```

Variable xPtr is then said to "point to" x



7.3 Pointer Operators II

∗ (indirection/dereferencing operator)

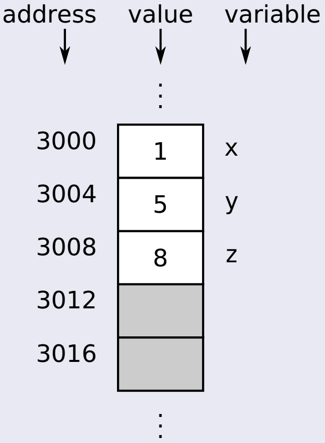
- ∗ not limited to pointer declaration, also used as operand
- ∗ provides access to memory/variable that its operand points to
- Access can be to read the value or to change the value

```
int x = 5; // assign 5 to x (direct reference)
int ∗xPtr; // declare an integer pointer
xPtr = &x; // assign the address of x to xPtr
printf("%d", ∗xPtr); // indirect reference to x, prints 5
∗xPtr = 7; // indirect reference, same effect as x = 7
```

  - Note: The names of pointer and variable are independent
- Dereferenced pointer (operand of ∗) must be an address (**lvalue**) (no constants)
- ∗ and & are inverses
  - They cancel each other out

7.3 Memory Concepts

Memory visualisation

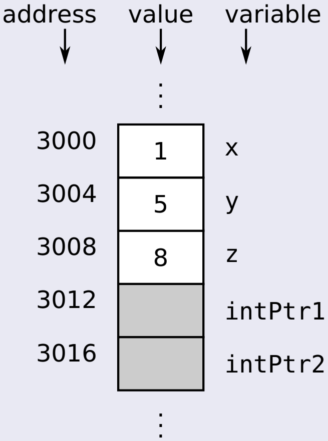


address	value	variable
	⋮	
3000	1	x
3004	5	y
3008	8	z
3012		
3016		
	⋮	

```
int x = 1;
int y = 5;
int z = 8;
```

7.3 Memory Concepts

Memory visualisation

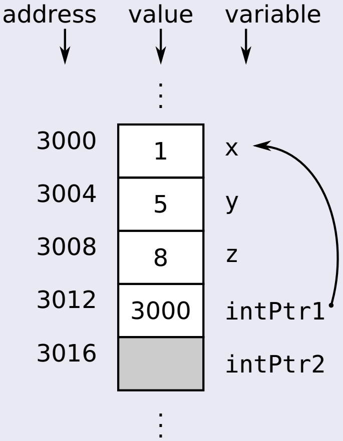


address	value	variable
	⋮	
3000	1	x
3004	5	y
3008	8	z
3012	▨	intPtr1
3016	▨	intPtr2
	⋮	

```
int x = 1;
int y = 5;
int z = 8;
int *intPtr1, *intPtr2;
```

7.3 Memory Concepts

Memory visualisation

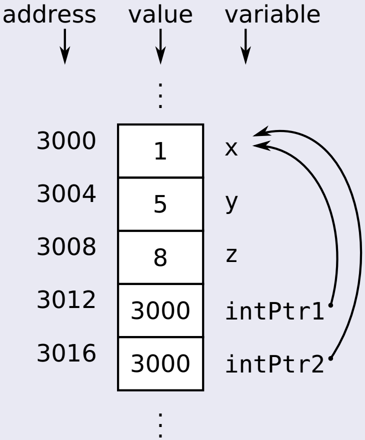


address	value	variable
⋮	⋮	
3000	1	x
3004	5	y
3008	8	z
3012	3000	intPtr1
3016	▨	intPtr2
⋮	⋮	

```
int x = 1;
int y = 5;
int z = 8;
int *intPtr1, *intPtr2;
intPtr1 = &x;
```

7.3 Memory Concepts

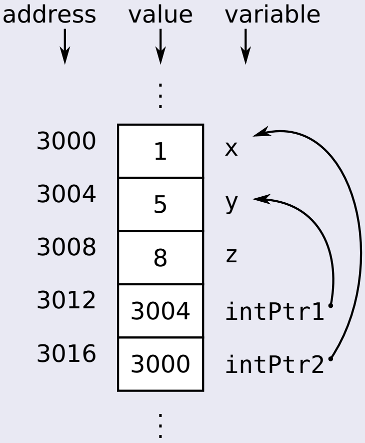
Memory visualisation



```
int x = 1;
int y = 5;
int z = 8;
int ∗intPtr1, ∗intPtr2;
intPtr1 = &x;
intPtr2 = intPtr1;
```

7.3 Memory Concepts

Memory visualisation

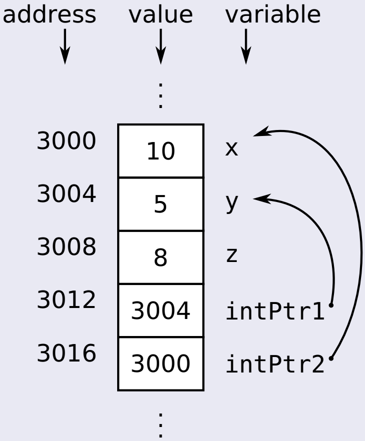


address	value	variable
	⋮	
3000	1	x
3004	5	y
3008	8	z
3012	3004	intPtr1
3016	3000	intPtr2
	⋮	

```
int x = 1;
int y = 5;
int z = 8;
int *intPtr1, *intPtr2;
intPtr1 = &x;
intPtr2 = intPtr1;
intPtr1 = &y;
```

7.3 Memory Concepts

Memory visualisation

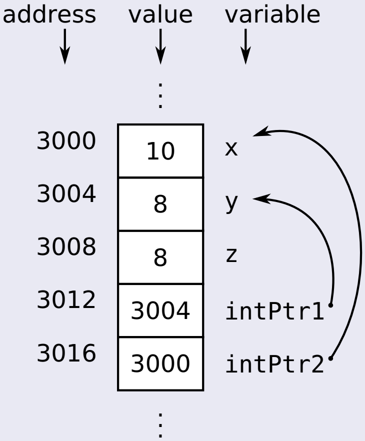


address	value	variable
	⋮	
3000	10	x
3004	5	y
3008	8	z
3012	3004	intPtr1
3016	3000	intPtr2
	⋮	

```c
int x = 1;
int y = 5;
int z = 8;
int *intPtr1, *intPtr2;
intPtr1 = &x;
intPtr2 = intPtr1;
intPtr1 = &y;
*intPtr2 = 10;
```

7.3 Memory Concepts

Memory visualisation

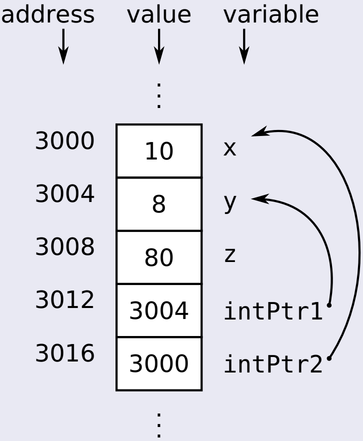


address	value	variable
⋮	⋮	
3000	10	x
3004	8	y
3008	8	z
3012	3004	intPtr1
3016	3000	intPtr2
	⋮	

```
int x = 1;
int y = 5;
int z = 8;
int *intPtr1, *intPtr2;
intPtr1 = &x;
intPtr2 = intPtr1;
intPtr1 = &y;
*intPtr2 = 10;
*intPtr1 = z;
```

7.3 Memory Concepts

Memory visualisation



```c
int x = 1;
int y = 5;
int z = 8;
int ∗intPtr1, ∗intPtr2;
intPtr1 = &x;
intPtr2 = intPtr1;
intPtr1 = &y;
∗intPtr2 = 10;
∗intPtr1 = z;
z = (∗intPtr1) ∗ (∗intPtr2);
```

```c
// Code to demonstrate pointer properties

#include <stdio.h>

int main()
{
   int a;        /* a is an integer */
   int *aPtr;    /* aPtr is a pointer to an integer */
   a = 7;
   aPtr = &a;    /* aPtr set to address of a */

   printf( "The address of a is %d"
           "\nThe value of aPtr is %d", (int) &a, (int) aPtr );
   printf( "\n\nThe value of a is %d"
           "\nThe value of *aPtr is %d", a, *aPtr );
   printf( "\n\nShowing that * and & are complements of "
           "each other\n&*aPtr = %d"
           "\n*&aPtr = %d\n", (int) &*aPtr, (int) *&aPtr );
   return 0;
}
```

Output

```
The address of a is  2686748
The value of aPtr is 2686748

The value of  a    is 7
The value of *aPtr is 7

Showing that * and & are complements of each other
&*aPtr = 2686748
*&aPtr = 2686748
```

7.3 Pointer Operators (cont...)

Operator precedence

	Operators							Associativity	Type
[]	()							left to right	highest
-	+	++	−−	!	*	&	(type)	right to left	unary
*	/	%						left to right	multiplicative
+	−							left to right	additive
<	<=	>	>=					left to right	relational
==	!=							left to right	equality
&&								left to right	logical and
||								left to right	logical or
?:								right to left	conditional
=	+=	−=	*=	/=	%=			right to left	assignment
,								left to right	comma

Perspective

Today

Pointers I

- Pointer definition
- Pointer declaration
- Pointer operations

Next lecture

Pointers II

- Passing pointers to functions

Homework

1. Study Sections 7.1-7.3 in Deitel & Deitel
2. Do Self Review Exercise 7.1 in Deitel & Deitel