

Week 3 Lecture 1

NWEN 241
Systems Programming

Alvin C. Valera

`alvin.valera@ecs.vuw.ac.nz`

Content

- Structures
- Pointers

Structures

Background

- Basic data types
 - int: integer ✓
 - char: character ✓
 - float: floating point number ✓
 - double: double-precision floating point number ✓
- Derived data types
 - Arrays ✓
 - Strings ✓
 - **Structures**

Structures

- A **struct** is a derived data type composed of members that are each basic or derived data types
- A single **struct** would store the data for one object. An array of **structs** would store the data for several objects
- A **struct** can be defined in several ways as illustrated in the following examples

Declaring a Structure

- Syntax of the structure type declaration:

```
struct structure_tag {  
    type1 member1;  
    type2 member2;  
    ...  
} variable_list;
```

- *structure_tag* specifies the name of the structure
- *structure_tag* and *variable_list* are optional
- If *structure_tag* is not specified, *variable_list* should be specified; otherwise, there is no way to declare variables using the unnamed structure type

Declaring a Structure

- Syntax of the structure type declaration:

```
struct structure_tag {  
    type1 member1;  
    type2 member2;  
    ...  
} variable_list;
```

- Structure members can be
 - Basic data types
 - Derived and user-defined types
 - *Pointers to basic, derived and user-defined data types*
 - *Function pointers*

Examples

- struct declaration that only defines a type:

```
struct student_info { // named struct
    char name [20];
    int student_id;
    int age;
}; // does not reserve any space
```

- struct declaration that defines a type **and** reserves storage for variables:

```
struct student_info { // named struct
    char name [20];
    int student_id;
    int age;
} s, t; // reserves space for s and t
```


Examples

- Declaring a variable `struct current_student`

```
struct student_info current_student;
```

- Above statement reserves space for:
 - 20 character array,
 - integer to store student ID, and
 - integer to store age

Examples

- Declaring array of structures to store information of enrolled students in a class

```
struct student_info nwen241class[250];
```

- Reserves space for 250 element array of records (structs) for students enrolled in NWEN241.

Creating New User Defined Types

- Instead of writing `struct student_info` every time we declare a variable, we can **define** it as a new data type

```
typedef struct {  
    char name [20];  
    int student_id;  
    int age;  
} StudentInfo;
```

- This makes `StudentInfo` a new user-defined type, and you can declare a variable as follows:

```
StudentInfo current_student;
```

New struct and Data Type

- If struct `student_info` has been previously defined, then we can create a new data type using `typedef` :

```
typedef struct student_info StudentInfo;
```

Initializing at Declaration (1)

- It is possible to initialize a struct at declaration

```
typedef struct {  
    char name [20];  
    int student_id;  
    int age;  
} StudentInfo;  
  
StudentInfo current_student =  
    { "John Doe", 12345, 18 };
```

- Order of initializer values should follow order of declaration

Initializing at Declaration (1)

- Partial initialization is also possible

```
typedef struct {  
    char name [20];  
    int student_id;  
    int age;  
} StudentInfo;  
  
StudentInfo current_student =  
    {"John Doe", 12345 };
```

- Remaining fields will be set to 0

Initializing at Declaration (2)

- It is possible to initialize certain fields of struct using **designated initialization**

```
typedef struct {  
    char name [20];  
    int student_id;  
    int age;  
} StudentInfo;
```

```
StudentInfo s1 = { .age = 18, .name = "John Doe" };  
// or StudentInfo s1 = { age: 18, name: "John Doe" };
```

- Initialization can be in any order

Accessing and Manipulating structs

- We can reference a component of a structure by the **direct component selection operator**, which is a **period**, e.g.

```
strcpy(student1.name, "John Smith");  
student1.age = 18;  
printf("%s is in age %d\n", student1.name, student1.age);
```

- The **direct component selection operator** has level 1 priority in the operator precedence
- Copying of an entire structure can be easily done by the assignment operator

```
student2 = student1;
```


Example – struct and typedef (1)

```
#include <stdio.h>
#include <string.h>

int main() {

    typedef struct student_info {
        char name[20];
        int student_id;
        int age;
    } StudentInfo;

    StudentInfo current_student;           // declare new variable using
                                           // new type StudentInfo
    struct student_info new_student;      // declare using struct format

    // do stuff – see next slide
```

Example – struct and typedef (2)

```
// declarations in previous slide

// initialize new student record
strcpy(new_student.name , "John Smith");
new_student.student_id = 300300300;
new_student.age = 22;

// copy new_student to current_student
current_student = new_student;

printf("Student name : %s\n", current_student.name);
printf("Student ID   : %.9d\n", current_student.student_id);
printf("Student Age   : %d\n", current_student.age);

}
```

Passing struct to Functions (1)

- Suppose there is a structure defined as follows

```
typedef struct {  
    char name[20];  
    double diameter;  
    int moons;  
    double orbit_time, rotation_time;  
} planet_t;
```

Passing struct to Functions (2)

- When a structure variable is passed as an input argument to a function, all its component values are **copied** into the local structure variable

```
1. /*
2.  * Displays with labels all components of a planet_t structure
3.  */
4. void
5. print_planet(planet_t pl) /* input - one planet structure */
6. {
7.     printf("%s\n", pl.name);
8.     printf("  Equatorial diameter: %.0f km\n", pl.diameter);
9.     printf("  Number of moons: %d\n", pl.moons);
10.    printf("  Time to complete one orbit of the sun: %.2f years\n",
11.           pl.orbit_time);
12.    printf("  Time to complete one rotation on axis: %.4f hours\n",
13.           pl.rotation_time);
14. }
```

Passing struct to Functions (3)

- Passing entire copy of a structure can be inefficient, especially for large structs
- There is a better way to pass structs to functions using pointers
 - To be discussed later

Pointers



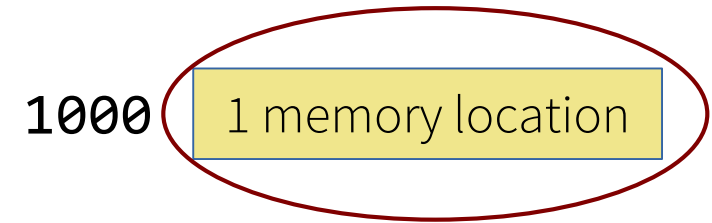
Memory Location

- All information accessible to a running computer program are stored somewhere in the computer's memory

Every *memory location* is identified by an **address**

1000	1 memory location
1001	
1002	
1003	
1004	
1005	
	...

Memory Location



- How big is 1 memory location?
 - It depends on the computer memory architecture

Word-addressable architecture:

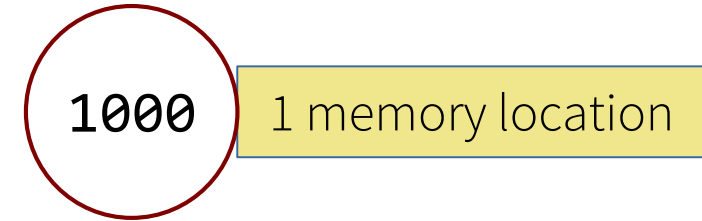
- Every memory location corresponds to one *word*

Byte-addressable architecture:

- Every memory location corresponds to one *byte*

Most computers today have byte-addressable memory

Memory Location



- **How big is the address?**
 - It depends on the number of bits used by CPU for addressing
- **Example:**
 - In a computer that uses 32 bits for addressing, an address has 32 bits
 - If the computer has byte-addressable memory, then the memory space is 2^{32} bytes = 4 gigabytes

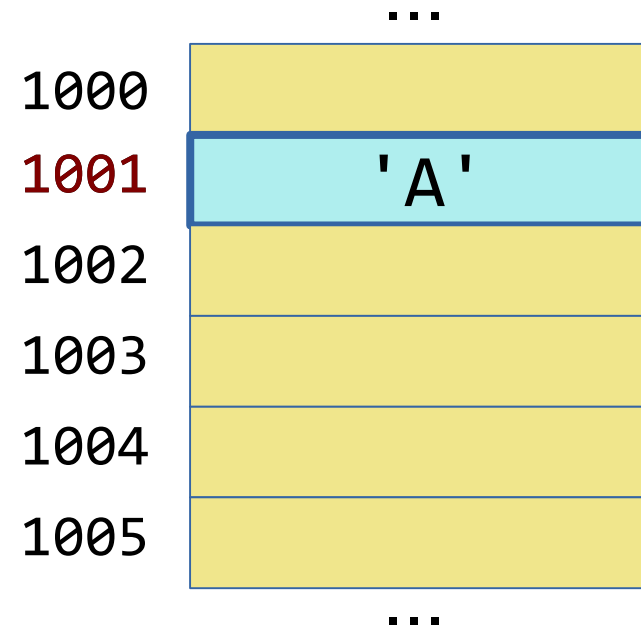
Memory Location and Variables

- A variable declaration allocates memory to store the value of the variable

```
char c = 'A';
```

Memory location 1001
contains value of
variable c

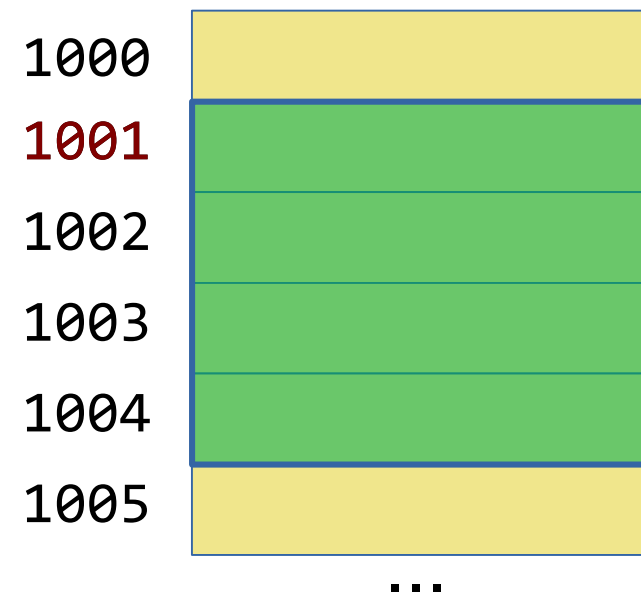
A variable **directly** references
a value



Memory Location and Variables

- In a byte-addressable computer, how do we address a data that occupies more than 1 byte, e.g., int, float or double?

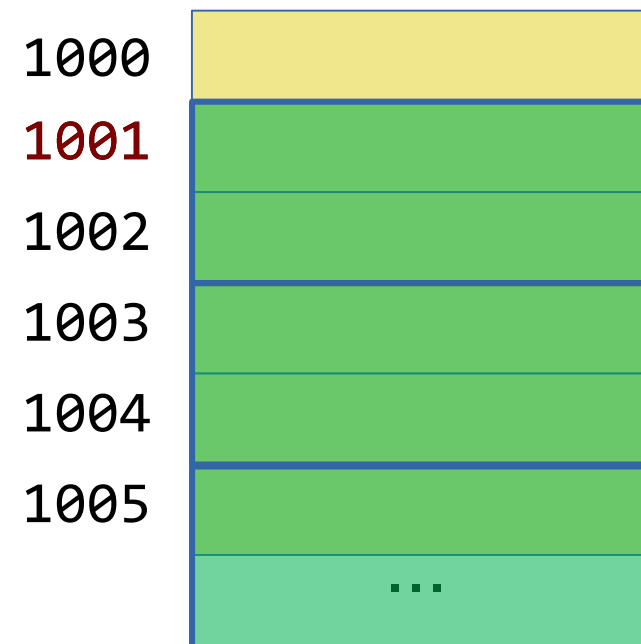
The address of a multi-byte data is the **starting address**



Memory Location and Variables

- In a byte-addressable computer, how do we address arrays?

The address of an array is the **starting address of the first element**



Memory Location and C

- C provides the ability to access specific memory locations, using **pointers**

*Pointers are variables that contain **memory addresses** as their values*

Variable vs Pointer

A variable **directly** references a value

A pointer **indirectly** references a value

Declaring a Pointer

- Pointers are typed based on the type of entity that they point to
 - To declare a pointer, use `*` preceding the variable name as in:

```
data_type *name;
```

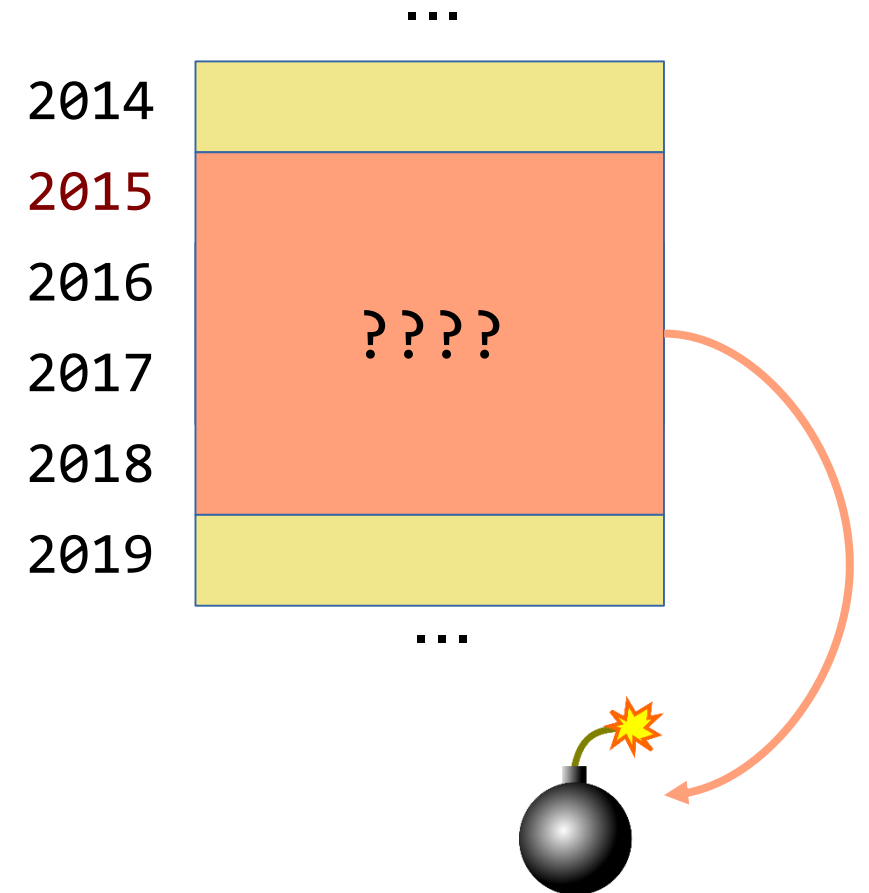
- Examples:

```
int *p;    // p is a pointer to an int
float *q;  // q is a float pointer
char *r;   // r is a char pointer
int *s[5]; // s is an array of 5 int pointers
```

What Happens in a Pointer Declaration?

```
int *p;
```

- Memory is allocated that can store an address
- The size of this space depends on the number of bits used for addressing
- The initial contents may be some 'rubbish' number
 - This means the pointer may point to arbitrary memory locations



Address Operator (&)

- The address (&) operator can be used in front of any variable
 - The operation will return the memory location of the variable

`&name`

name can be any ordinary variable or even a pointer variable

- Example:

```
int a, *x;  
x = &a;  
/* x variable contains address of a, i.e.,  
x points to variable a */
```


Indirection Operator (*)

- A pointer variable contains a memory address
- To refer to the *contents* of the variable that the pointer points to, we use indirection operator

*name

name is a pointer variable

- Example:

```
int a = 100, b, *x;  
x = &a;  
b = *x;  
/* b will be assigned the content pointed  
to by x, which is 100 */
```

Next Lecture

- More Pointers