Announcements

- Homework assignment due Friday
- Read chapters 5-7 in book
Lecture 13: Overview

- Data Structures
  - Structures
    - Declaration and definition
    - Instantiation and initialization
    - Member access
  - Unions
    - Declaration and definition
    - Member access
  - Enumerators
    - Declaration and definition
  - Type definitions

- Data Structures
  - Memory organization

- Objects in memory

- Pointers
  - Pointer definition
  - Pointer operators
  - Pointer dereferencing
Data Structures

- Structures (aka. records): `struct`
  - User-defined, composite data type
    - Type is a composition of (different) sub-types
  - Fixed set of members
    - Names and types of members are fixed at structure definition
  - Member access by name
    - Member-access operator: `structure_name.member_name`

- Example:

```c
struct S { int i; float f; } s1, s2;

s1.i = 42;    /* access to members */
s1.f = 3.1415;
s2 = s1;      /* assignment */
s1.i = s1.i + 2*s2.i;
```
Data Structures

- Structure Declaration
  - Declaration of a user-defined data type
- Structure Definition
  - Definition of structure members and their type
- Structure Instantiation and Initialization
  - Definition of a variable of structure type
  -Initializer list defines initial values of members
- Example:

```c
struct Student;          /* declaration */
struct Student           /* definition */
{ int   ID;              /* members */
    char  Name[40];
    char  Grade;
};
struct Student Jane =    /* instantiation */
{1001, "Jane Doe", 'A'}; /* initialization */
```
Data Structures

- **Structure Access**
  - Members are accessed by their name
  - Member-access operator .
- **Example:**

```c
struct Student {
    int ID;
    char Name[40];
    char Grade;
};

struct Student Jane = {1001, "Jane Doe", 'A'};

void PrintStudent(struct Student s) {
    printf("ID:    %d\n", s.ID);
    printf("Name:  %s\n", s.Name);
    printf("Grade: %c\n", s.Grade);
}
```

Jane

<table>
<thead>
<tr>
<th>ID</th>
<th>1001</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>&quot;Jane Doe&quot;</td>
</tr>
<tr>
<td>Grade</td>
<td>'A'</td>
</tr>
</tbody>
</table>

ID: 1001
Name: Jane Doe
Grade: A
Data Structures

- Unions: `union`
  - User-defined, composite data type
    - Type is a composition of (different) sub-types
  - Fixed set of mutually exclusive members
    - Names and types of members are fixed at union definition
  - Member access by name
    - Member-access operator: `union_name.member_name`
  - *Only one member may be used at a time!*
    - *All members share the same location in memory!*

- Example:

```c
union U { int i; float f; } u1, u2;
u1.i = 42;     /* access to members */
u2.f = 3.1415;
u1.f = u2.f;   /* destroys u1.i! */
```
Data Structures

- Union Declaration
  - Declaration of a user-defined data type
- Union Definition
  - Definition of union members and their type
- Union Instantiation and Initialization
  - Definition of a variable of union type
  - \textit{Single} initializer defines value of \textit{first} member
- Example:

```c
union HeightOfTriangle; /* declaration */
union HeightOfTriangle /* definition */
{ int   Height;          /* members */
  int   LengthOfSideA;
  float AngleBeta;
};
union HeightOfTriangle H /* instantiation */
= { 42 };                /* initialization */
```
Data Structures

- **Union Access**
  - Members are accessed by their name
  - Member-access operator .
- **Example:**

```c
union HeightOfTriangle
{ int Height;
  int SideA;
  float Beta;
};
union HeightOfTriangle t1, t2, t3 = { 42 };
```

- Example values:
  - `t1`: Height = 0, SideA = 0, Beta = 42
  - `t2`: Height = 0, SideA = 0, Beta = 42
  - `t3`: Height = 0, SideA = 42, Beta = 42
Data Structures

• Union Access
  • Members are accessed by their name
  • Member-access operator .

• Example:

```c
union HeightOfTriangle
{ int   Height;
  int   SideA;
  float Beta;
};
union HeightOfTriangle t1, t2, t3 = { 42 };
void SetHeight(void)
{
  t1.Height = 10;
  t2.SideA = t1.Height / 2;
  t3.Beta = 90.0;
}
```

```
<table>
<thead>
<tr>
<th></th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Height</td>
<td>10.0</td>
<td>5.0</td>
<td>90.0</td>
</tr>
<tr>
<td>SideA</td>
<td></td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>Beta</td>
<td></td>
<td>5.0</td>
<td>90.0</td>
</tr>
</tbody>
</table>
```
Data Structures

- Enumerators: `enum`
  - User-defined data type
    - Members are an enumeration of integral constants
  - Fixed set of members
    - Names and values of members are fixed at enumerator definition
  - Members are constants
    - Member values cannot be changed after definition
- Example:

```cpp
enum E { red, yellow, green }; enum E LightNS, LightEW;

LightEW = green;    /* assignment */
if (LightNS == green)    /* comparison */
    { LightEW = red; }
```
Data Structures

- Enumerator Declaration
  - Declaration of a user-defined data type
- Enumerator Definition
  - Definition of enumerator members and their value
- Enumerator Instantiation and Initialization
  - Definition of a variable of enumerator type
  - Initializer should be one member of the enumerator
- Example:

```c
enum Weekday;            /* declaration */
enum Weekday             /* definition */
{ Monday, Tuesday,       /* members */
  Wednesday, Thursday,
  Friday, Saturday, Sunday;
};
enum Weekday Today       /* instantiation */
= Wednesday;             /* initialization */
```
Data Structures

• Enumerator Values
  • Enumerator values are integer constants
  • By default, enumerator values start at 0 and are incremented by 1 for each following member

• Example:

```c
enum Weekday { Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday; }
enum Weekday Today = Wednesday;
void PrintWeekday(enum Weekday d)
{
    printf("Day: %d\n", d);
}
```
Data Structures

- Enumerator Values
  - Enumerator values are integer constants
  - By default, enumerator values start at 0 and are incremented by 1 for each following member
  - Specific enumerator values may be defined by the user
- Example:

```c
enum Weekday { Monday = 1, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday;
};
enum Weekday Today = Wednesday;
void PrintWeekday(enum Weekday d)
{
    printf("Day: %d\n", d);
}
```
Data Structures

- Enumerator Values
  - Enumerator values are integer constants
  - By default, enumerator values start at 0 and are incremented by 1 for each following member
  - Specific enumerator values may be defined by the user
- Example:

```c
enum Weekday { Monday = 2,
                Tuesday,
                Wednesday,
                Thursday,
                Friday,
                Saturday,
                Sunday = 1;  
};

enum Weekday Today = Wednesday;

void PrintWeekday(enum Weekday d)
{
    printf("Day: %d\n", d);
}
```
Data Structures

- Type definitions: `typedef`
  - A `typedef` can be defined as an alias type for another type
  - A `typedef` definition follows the same rules as a variable definition
  - Type definitions are usually used to abbreviate access to user-defined types
- Examples:

```c
typedef long MyInteger;

typedef enum Weekday Day;
Day Today;

typedef struct Student Scholar;
Scholar Jane, John;
```
Memory Organization

- Memory Segmentation
  - typical (virtual) memory layout on processor with 4-byte words and 1 GB of memory
- Stack
  - grows and shrinks dynamically
  - function call hierarchy
  - stack frames with local variables
- Heap
  - “free” storage
  - dynamic allocation by the user
- Data segment
  - global (and static) variables
- Program segment
  - stores binary program code
- Reserved area for operating system
Memory Organization

- Memory Segmentation
  - typical (virtual) memory layout on processor with 4-byte words and 1 GB of memory

- Memory errors
  - *Out of memory*
    - Stack and heap collide
  - *Segmentation fault*
    - access outside allocated segments
    - e.g. access to segment reserved for OS
  - *Bus error*
    - mis-aligned word access
    - e.g. word access to an address that is not divisible by 4
Objects in Memory

- Data in memory is organized as a set of objects
- Every object has ...
  - ... a **type** (e.g. `int`, `double`, `char[5]`)
    - type is known to the compiler at compile time
  - ... a **value** (e.g. `42`, `3.1415`, "text")
    - value is used for computation of expressions
  - ... a **size** (number of bytes in the memory)
    - in C, the `sizeof` operator returns the size of a variable or type
  - ... a **location** (address in the memory)
    - in C, the "address-of" operator (`&`) returns the address of an object
- Variables ...
  - ... serve as identifiers for objects
  - ... are bound to objects
  - ... give objects a name
Objects in Memory

- Example: Variable values, addresses, and sizes

```c
int x = 42;
int y = 13;
char s[] = "Hello World!";

printf("Value of x is %d.\n", x);
printf("Address of x is %p.\n", &x);
printf("Size of x is %u.\n", sizeof(x));
printf("Value of y is %d.\n", y);
printf("Address of y is %p.\n", &y);
printf("Size of y is %u.\n", sizeof(y));
printf("Value of s is %s.\n", s);
printf("Address of s is %p.\n", &s);
printf("Size of s is %u.\n", sizeof(s));
printf("Value of s[1] is %c.\n", s[1]);
printf("Address of s[1] is %p.\n", &s[1]);
printf("Size of s[1] is %u.\n", sizeof(s[1]));
```
Objects in Memory

- Example: Variable values, addresses, and sizes

```c
int x = 42;
int y = 13;
char s[] = "Hello World!";
...
```

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ffbefa4c</td>
<td>42</td>
</tr>
<tr>
<td>ffbefa48</td>
<td>13</td>
</tr>
<tr>
<td>ffbefa44</td>
<td>0</td>
</tr>
<tr>
<td>ffbefa40</td>
<td>'r' 'l' 'd' '!'</td>
</tr>
<tr>
<td>ffbefa3c</td>
<td>'o' ' ' 'W' 'o'</td>
</tr>
<tr>
<td>ffbefa38</td>
<td>'H' 'e' 'l' 'l'</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
• **Pointers** are variables whose values are *addresses*
  • The “address-of” operator (&) returns a pointer!
• Pointer Definition
  • The unary * operator indicates a pointer type in a definition

```c
int x = 42; /* regular integer variable */
int *p; /* pointer to an integer */
```

• Pointer initialization or assignment
  • A pointer may be set to the “address-of” another variable

```c
p = &x; /* p points to x */
```

• A pointer may be set to 0 (points to no object)

```c
p = 0; /* p points to no object */
```

• A pointer may be set to **NULL** (points to “NULL” object)

```c
#include <stdio.h> /* defines NULL as 0 */
p = NULL; /* p points to no object */
```
Pointers

• Pointer Dereferencing
  • The unary * operator dereferences a pointer to the value it points to (“content-of” operator)

```c
#include <stdio.h>
int x = 42; /* regular integer variable */
int *p = NULL; /* pointer to an integer */
```

```
P 0
x 42
```
Pointers

- Pointer Dereferencing
  - The unary * operator dereferences a pointer to the value it points to ("content-of" operator)

```c
#include <stdio.h>

int x = 42;  /* regular integer variable */
int *p = NULL;  /* pointer to an integer */
p = &x;  /* make p point to x */
```

![Diagram showing pointer dereferencing](image-url)
Pointers

- Pointer Dereferencing
  - The unary * operator dereferences a pointer to the value it points to ("content-of" operator)

```c
#include <stdio.h>
int x = 42; /* regular integer variable */
int *p = NULL; /* pointer to an integer */
p = &x; /* make p point to x */
printf("x is %d, content of p is %d\n", x, *p);
```

```
x is 42, content of p is 42
```
Pointers

- Pointer Dereferencing
  - The unary * operator dereferences a pointer to the value it points to ("content-of" operator)

```c
#include <stdio.h>

int x = 42;    /* regular integer variable */
int *p = NULL; /* pointer to an integer */
p = &x;        /* make p point to x */
printf("x is %d, content of p is %d\n", x, *p);
*p = 2 * *p;   /* multiply content of p by 2 */
printf("x is %d, content of p is %d\n", x, *p);
```

- x is 42, content of p is 42
- x is 84, content of p is 84
Pointers

- **Pointer Dereferencing**
  - The `->` operator dereferences a pointer to a structure to the content of a structure member

```c
struct Student
{
    int ID;
    char Name[40];
    char Grade;
};

struct Student Jane = {1001, "Jane Doe", 'A'};
struct Student *p = &Jane;

void PrintStudent(void)
{
    printf("ID:    %d\n", p->ID);
    printf("Name:  %s\n", p->Name);
    printf("Grade: %c\n", p->Grade);
}
```