Announcements

• Homework assignment due Friday
• Read chapters 5-7 in book
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
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
  - *Single* initializer defines value of *first* member
- Example:

```c
union HeightOfTriangle; /* declaration */
union HeightOfTriangle /* definition */
{ int Height; /* members */
  int LengthOfSideA;
  float AngleBeta;
};

union HeightOfTriangle H /* instantiation */
= { 42 }; /* initialization */
```
• 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 };
```
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;
}
```
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:

```c
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:

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

Today  
Wednesday  
Day: 2
**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);
}
```

Today: **Wednesday**

Day: 3
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);
}
```

Today

Wednesday

Day: 4
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;
```
Basic Computer Architecture

- Essential Computer Components
  - Central Processing Unit (CPU)
    - e.g. Intel Pentium, Motorola PowerPC, Sun SPARC, ...
  - Random Access Memory (RAM)
    - storage for program and data, read and write access
  - Read Only Memory (ROM)
    - fixed storage for basic input/output system (BIOS)
  - I/O Units
    - Input/output units connecting to peripherals

![Diagram of basic computer architecture with components connected by busses: Clock, CPU, RAM, ROM, I/O, Data Bus, Address Bus, I/O Busses]
Programs and data in a computer are represented in binary format:

- 1 bit (binary digit), 2 possible values:
  - 0 (false, “no”, current off, “empty”, ...)
  - 1 (true, “yes”, current on, “solid”, ...)
- 1 byte = 8 bits \( (2^8 = 256 \text{ values}) \)
  - in C, type `char` equals one byte
- 1 word = 4 bytes \( (2^{32} = 4294967296 \text{ values}) \)
  - in C, type `int` equals one word
- Memory size is measured in Bytes:
  - 1 KB = 1024 byte = 1 “kilo byte”
  - 1 MB = 1024*1024 byte = 1 “mega byte”
  - 1 GB = 1024*1024*1024 byte = 1 “giga byte”

(*architecture dependent!*)
Binary Data Representation

• Memory is composed of addressable bytes
  • Example:
    1 KB of memory
  • What is the value at address 7?

\[
\begin{align*}
7 & \quad \square \quad \square \quad \square \quad \square \quad \square \quad \square \quad \square \\
6 & \quad 5 & \quad 4 & \quad 3 & \quad 2 & \quad 1 & \quad 0 \\
\end{align*}
\]

\[
\begin{align*}
= \ 0 \times 2^7 & + 1 \times 2^6 & + 0 \times 2^5 & + 0 \times 2^4 \\
+ 1 \times 2^3 & + 1 \times 2^2 & + 0 \times 2^1 & + 1 \times 2^0 \\
= \ 0 \times 128 & + 1 \times 64 & + 0 \times 32 & + 0 \times 16 \\
+ 1 \times 8 & + 1 \times 4 & + 0 \times 2 & + 1 \times 1 \\
= \ 64 & + 8 & + 4 & + 1 \\
= \ 77 \\
\end{align*}
\]
Binary Data Representation

• Number Systems
  • DEC: Decimal numbers
    • Base 10, digits 0, 1, 2, 3, ..., 9
    • e.g. 157 = 1*10^2 + 5*10^1 + 7*10^0
  • BIN: Binary numbers
    • Base 2, digits 0, 1
    • e.g. 10011101_2 = 1*2^7 + 0*2^6 + ... + 1*2^0
  • OCT: Octal numbers
    • Base 8, digits 0, 1, 2, 3, ..., 7
    • e.g. 235_8 = 2*8^2 + 3*8^1 + 5*8^0
  • HEX: Hexadecimal numbers
    • Base 16, digits 0, 1, 2, 3, ..., 9, A, B, C, ..., F
    • e.g. 9D_{16} = 9*16^1 + 13*16^0
Binary Data Representation

- **Number Systems**

<table>
<thead>
<tr>
<th>DEC</th>
<th>BIN</th>
<th>OCT</th>
<th>HEX</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
<td>11</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>1010</td>
<td>12</td>
<td>A</td>
</tr>
<tr>
<td>11</td>
<td>1011</td>
<td>13</td>
<td>B</td>
</tr>
<tr>
<td>12</td>
<td>1100</td>
<td>14</td>
<td>C</td>
</tr>
<tr>
<td>13</td>
<td>1101</td>
<td>15</td>
<td>D</td>
</tr>
<tr>
<td>14</td>
<td>1110</td>
<td>16</td>
<td>E</td>
</tr>
<tr>
<td>15</td>
<td>1111</td>
<td>17</td>
<td>F</td>
</tr>
</tbody>
</table>
**Binary Data Representation**

- **Number Systems (signed vs. unsigned)**

<table>
<thead>
<tr>
<th>SDEC</th>
<th>UDEC</th>
<th>BIN</th>
<th>OCT</th>
<th>HEX</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0001</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0010</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0011</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>0100</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>0101</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>0110</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>0111</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>-8</td>
<td>8</td>
<td>1000</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>-7</td>
<td>9</td>
<td>1001</td>
<td>11</td>
<td>9</td>
</tr>
<tr>
<td>-6</td>
<td>10</td>
<td>1010</td>
<td>12</td>
<td>A</td>
</tr>
<tr>
<td>-5</td>
<td>11</td>
<td>1011</td>
<td>13</td>
<td>B</td>
</tr>
<tr>
<td>-4</td>
<td>12</td>
<td>1100</td>
<td>14</td>
<td>C</td>
</tr>
<tr>
<td>-3</td>
<td>13</td>
<td>1101</td>
<td>15</td>
<td>D</td>
</tr>
<tr>
<td>-2</td>
<td>14</td>
<td>1110</td>
<td>16</td>
<td>E</td>
</tr>
<tr>
<td>-1</td>
<td>15</td>
<td>1111</td>
<td>17</td>
<td>F</td>
</tr>
</tbody>
</table>
### Binary Data Representation

- **Number Systems**
  - **Signed representation:** *two’s complement*
    - to obtain the negative of any number in binary representation,
      - ... invert all bits,
      - ... and add 1

- **Example: 4-bit two’s complement**

<table>
<thead>
<tr>
<th>SDEC</th>
<th>UDEC</th>
<th>BIN</th>
<th>OCT</th>
<th>HEX</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>0111</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>−8</td>
<td>9</td>
<td>1000</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>−7</td>
<td></td>
<td>1001</td>
<td>11</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

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

<table>
<thead>
<tr>
<th>Value</th>
<th>Address</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>ffbefa4c</td>
<td>4</td>
</tr>
<tr>
<td>y</td>
<td>ffbefa48</td>
<td>4</td>
</tr>
<tr>
<td>s</td>
<td>ffbefa38</td>
<td>13</td>
</tr>
<tr>
<td>s[1]</td>
<td>ffbefa39</td>
<td>1</td>
</tr>
</tbody>
</table>

Stack:

```
ffbefa4c
ffbefa48
ffbefa44
ffbefa40
ffbefa3c
ffbefa38
...```
Pointers

- 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  x
0  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);
```

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);
}
```
Pointers

- **Pointer Arithmetic**
  - Pointers pointing into arrays may be ... 
    - ... incremented to point to the next array element
    - ... decremented to point to the previous array element

```c
int x[5] = {10, 20, 30, 40, 50}; /* array of 5 integers */
int *p; /* pointer to integer */
p = &x[1]; /* point p to x[1] */
printf("%d, ", *p); /* print content of p */
p++; /* increment p by 1 */
p--; /* decrement p by 1 */
printf("%d, ", *p); /* print content of p */
p += 2; /* increment p by 2 */
printf("%d, ", *p); /* print content of p */

20, 30, 20, 40,
```
Pointers

• Pointer Comparison
  • Pointers may be compared for equality
    • operators == and != are useful to determine identity
    • operators <, <=, >=, and > are not applicable

```c
int x[5] = {10,20,10,20,10}; /* array of 5 integers */
int *p1, *p2; /* pointers to integer */
p1 = &x[1]; p2 = &x[3]; /* point to x[1], x[3] */

if (p1 == p2)
  { printf("p1 and p2 are identical!\n"); }
if (*p1 == *p2)
  { printf("Contents of p1 and p2 are the same!\n"); }
```

Contents of p1 and p2 are the same!
Pointers

• Pointer Comparison
  • Pointers may be compared for equality
    • operators == and != are useful to determine identity
    • operators <, <=, >=, and > are not applicable

```c
int x[5] = {10,20,10,20,10}; /* array of 5 integers */
int *p1, *p2;            /* pointers to integer */
p1 = &x[1]; p2 = &x[3];   /* point to x[1], x[3] */
p1 += 2;                 /* increment p1 by 2 */
if (p1 == p2)
  { printf("p1 and p2 are identical!\n"); }
if (*p1 == *p2)
  { printf("Contents of p1 and p2 are the same!\n"); }
```

p1 and p2 are identical!
Contents of p1 and p2 are the same!
Pointers

• String Operations using Pointers
  • Example: String length

```c
int Length(char *s)
{
    int l = 0;
    char *p = s;

    while(*p != 0)
    {
        p++;
        l++;
    }

    return l;
}
```

```
char s1[] = "ABC";
char s2[] = "Hello World!";

printf("Length of %s is %d\n", s1, Length(&s1[0]));
printf("Length of %s is %d\n", s2, Length(&s2[0]));
```

Length of ABC is 3
Length of Hello World! is 12
Pointers

- String Operations using Pointers
  - Example: String length

```c
int Length(char *s)
{
    int l = 0;
    char *p = s;
    while(*p != 0)
    {
        p++;
        l++;
    }
    return l;
}
```

```
char s1[] = "ABC";
char s2[] = "Hello World!";

printf("Length of %s is %d\n", s1, Length(&s1[0]));
printf("Length of %s is %d\n", s2, Length(s2));
```

- Array and pointer types are equivalent
  - `s2` is an array, but can be passed as a pointer argument
  - Character array `s2` is same as character pointer `&s2[0]`
Pointers

• String Operations using Pointers
  • Example: String length

```c
int Length(char *s) {
    int l = 0;
    char *p = s;
    while(*p != 0) {
        p++;
        l++;
    }
    return l;
}
```

```c
char s1[] = "ABC";
char *s2 = "Hello World!";

printf("Length of %s is %d\n", s1, Length(s1));
printf("Length of %s is %d\n", s2, Length(s2));
```

• Array and pointer types are equivalent
  • `s1` is an array of characters, `s2` is a pointer to character
  • Both `s1` and `s2` can be passed to character pointer `s`
Pointers

- String Operations using Pointers
  - Example: String length

```
int Length(char s[]) {
    int l = 0;
    char *p = s;
    while(*p != 0) {
        p++;
        l++;
    }
    return l;
}
```

```
char s1[] = "ABC";
char *s2 = "Hello World!";

printf("Length of %s is %d\n", s1, Length(s1));
printf("Length of %s is %d\n", s2, Length(s2));
```

- Array and pointer types are equivalent
  - \texttt{s1} is an array of characters, \texttt{s2} is a pointer to character
  - Both \texttt{s1} and \texttt{s2} can be passed to character array \texttt{s}
Pointers

- String Operations using Pointers
  - Example: String copy

```c
void Copy(
    char *Dst,
    char *Src)
{
    do{
        *Dst = *Src;
        Dst++;
    } while(*Src++);
}
```

```c
char s1[] = "ABC";
char s2[] = "Hello World!";

printf("s1 is %s, s2 is %s\n", s1, s2);
Copy(s2, s1);
printf("s1 is %s, s2 is %s\n", s1, s2);
```

- Passing pointers as arguments to functions
  - Function can modify caller data by pointer dereferencing
    - Passing pointers = Pass by reference!
Pointers

- String Operations using Pointers
  - Example: String copy

```c
void Copy(
  char *Dst,
  const char *Src)
{
  do{
    *Dst = *Src;
    Dst++;
  } while(*Src++);
}
```

```c
char s1[] = "ABC";
char s2[] = "Hello World!";

printf("s1 is %s, s2 is %s\n", s1, s2);
Copy(s2, s1);  
printf("s1 is %s, s2 is %s\n", s1, s2);
```

- Passing pointers as arguments to functions
  - Function can modify caller data by pointer dereferencing
  - Type qualifier `const`: Modification by pointer dereferencing *not* allowed!
Pointers

- String Operations using Pointers
  - Example: String copy

```c
void Copy(
    const char *Dst,
    const char *Src)
{
    do{
        *Dst = *Src;
        Dst++;
        while(*Src++);
    }
}
```

```c
char s1[] = “ABC”;
char s2[] = “Hello World!”;

printf(“s1 is %s, s2 is %s\n”,
        s1, s2);
Copy(s2, s1);
printf(“s1 is %s, s2 is %s\n”,
        s1, s2);
```

Error!
Write access to 
const data!

- Passing pointers as arguments to functions
  - Function can modify caller data by pointer dereferencing
  - Type qualifier `const`:
    Modification by pointer dereferencing `not` allowed!
Allocating memory in the heap

- `malloc` (memory allocate) function
  - need to include `stdlib.h` to use
  - `int * ptr = (int *)malloc(sizeof(int));`
    - takes in a size (in bytes)
    - returns a pointer to a block of memory of the specified size
    - or NULL is there is an error
Freeing memory in the heap

• when you’re done with the memory use free to return it
  • free(ptr);
  • free takes in a pointer to beginning of memory block and returns that block back to the operating system
Malloc & Free

int * int_ptr=(int*)malloc(sizeof(int));
*int_ptr=1;
free(int_ptr);
Translation Units

• Introduction
  • C compilation process is a sequence of phases
    • Preprocessing (handle # directives)
    • Scanning and parsing (generate internal data structure)
    • Instruction generation (emit stream of CPU instructions)
    • Assembly (generate binary object file)
    • Linking (combine objects into executable file)
  • C compiler consists of separate components
    • Preprocessor (processes # directives)
    • Compiler (compiles and assembles code)
    • Linker (processes object files and libraries)
Translation Units

- Compilation Phases
  - Source code
    - Program file
    - Header file(s)
  - Preprocessed file

- Source code
  - Program file
  - Header file(s)

- Preprocessor
  - Prg.c
  - Prg.h

- Compiler
  - Prg.i

- Linker
  - Prg.o
  - libc.a
  - libm.a

- Executable file
  - Prg
Translation Units

• Source files
  • Header files: Program.h
    • Inclusion of required header files
    • Definitions of exported constants
    • Declarations of exported global variables
    • Declarations of exported functions
  • Program files: Program.c
    • Inclusion of required header files
    • Declaration and definition of local variables
    • Declaration and definition of local functions
    • Definitions of exported global variables
    • Definitions of exported functions
Translation Units

- **Object files**
  - `Program.o`
    - Compiled object code of source file `Program.c`
    - Use option `-c` in GNU compiler call to create object files
      
      ```
      gcc -c Program.c -o Program.o -Wall -ansi
      ```
  - Library.a
    - Archive of compiled object files

- **Executable file**
  - `Program`
    - Object files and libraries linked together into a complete file ready for execution
    - GNU compiler recognizes object files by .o suffix, so object files and libraries require no special option
      
      ```
      gcc Program.o -lc -lm -o Program
      ```
Translation Units

• Multiple Translation Units
  • C programs can be partitioned into multiple translation units, aka. *modules*
  • Modules typically consist of
    • Module header file (file suffix `.h`)
    • Module program file (file suffix `.c`)
    • Module object file (file suffix `.o`)
  • Modules are *linked* together
    • Linker combines object files and required libraries into an executable file
      • `gcc Program.o Mod1.o Mod2.o -lc -lm -o Program`
Translation Units

• Multiple Translation Units

M1.c  M1.h  M2.c  M2.h  Prg.c  Prg.h  stdio.h  math.h

Preprocessor  Preprocessor  Preprocessor

M1.i  M2.i  Prg.i

Compiler  Compiler  Compiler

M1.o  M2.o  Prg.o

Libc.a  Libm.a

Linker  Prg