Applied Programming

Memory and Pointers

Memory and Pointers

- Computer Memory
  - Stack and Heap
- C Pointers

Computer Memory Model

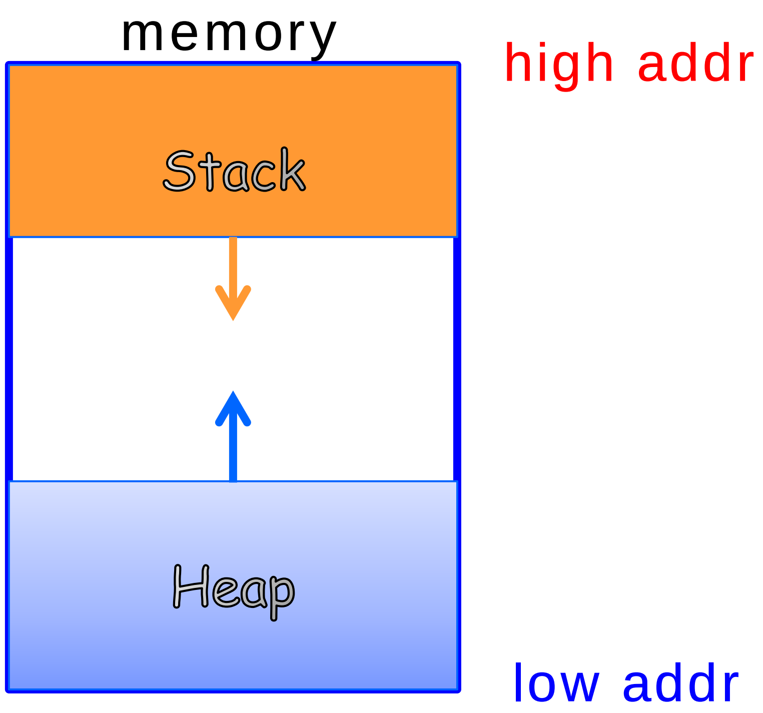
- **Stack Memory**
  - Used for ***local variables*** (from functions) and related book-keeping (parameter passing, return addresses)
  - LIFO structure:
    - Function variables pushed onto stack when called
    - Function variables popped off stack on return

- **Heap Memory**
  - Memory available for ***dynamically allocated variables*** (**`malloc`**)
  - Once (heap) memory allocated it remains taken until it is **`freed`** or the program terminates.

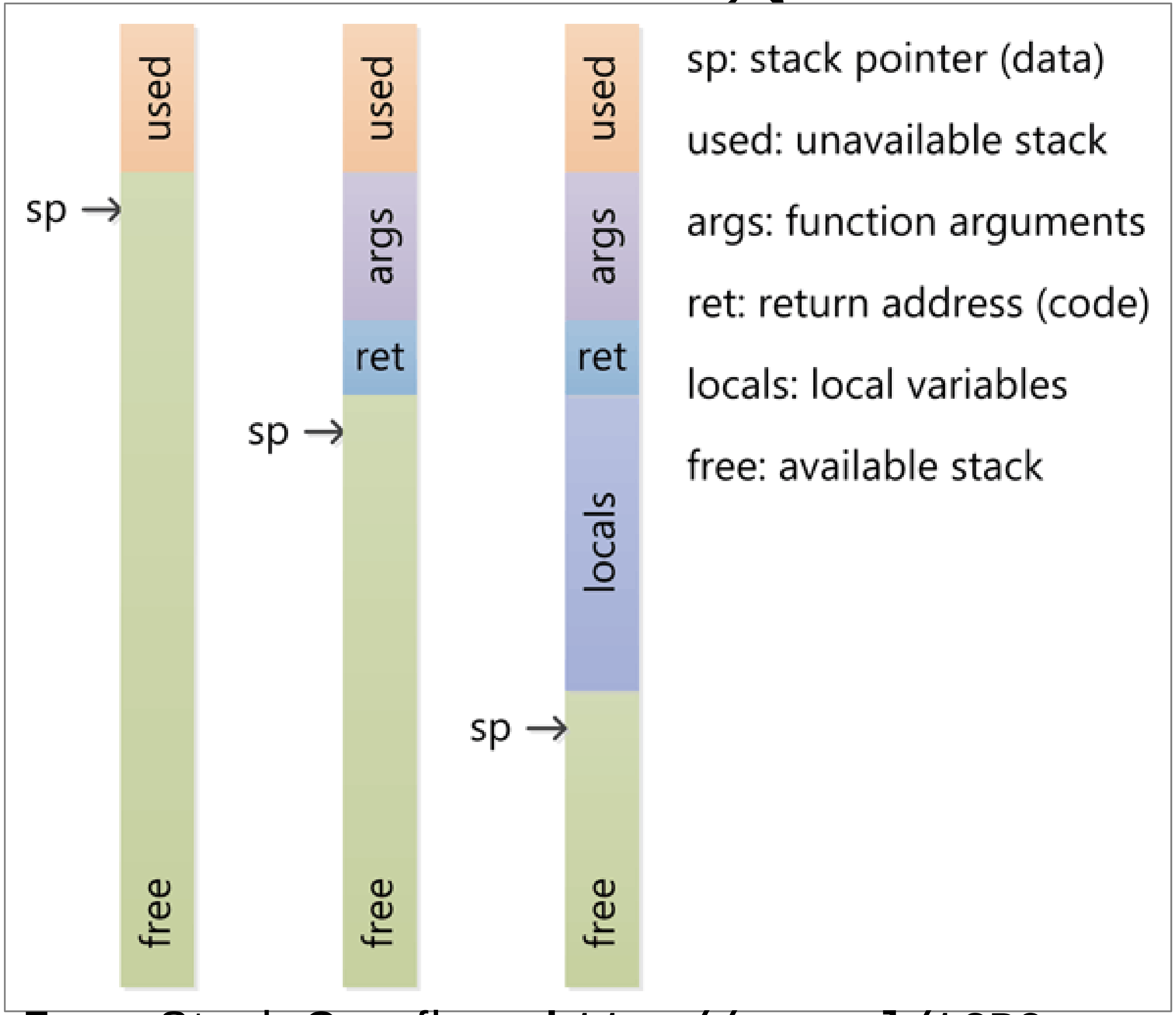
Memory Layout

- Where is it located (mapped)?
  - Depends on Architecture
    - **`x86/x86_46`**: Stack Grows Down
    - **`ARM`**: selectable

- In general:
  - Stack grows down (from upper memory addresses)
  - Heap grows up (from lower memory) addresses)



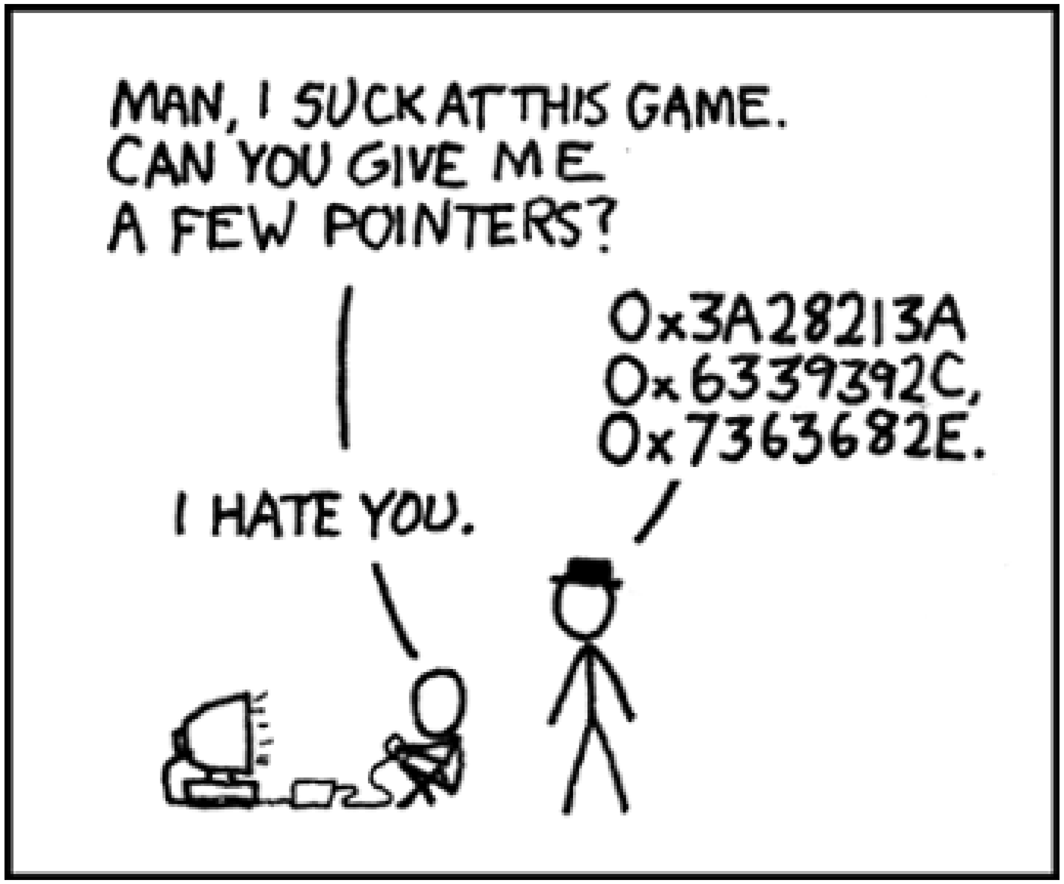
Stack Diagram



From Stack Overflow: **`http://goo.gl/t2PQo`**

Pointers and Addresses

- To understand ***dynamic memory allocation in C*** we need to introduce the concept of ***pointer***



From xkcd: http://xkcd.com/138 (Creative Commons)

Pointers Overview

- Pointers
  - A ***pointer*** is a variable that stores the ***memory address of another variable***
  - *each variable has an address and a value*
- What does not have an address ?
  - register variables
  - constants/literals/preprocessor defines
  - Expressions (unless result is a variable)
- How do we find the address of a variable, or a function?

& and * Operators

- Pointer Operators in C

  There are **two unary operators** used with pointers:

  1. **Address of operator** (**&**)

     - returns the ***address*** of a variable

     ```
     &foo; /* address of variable foo */
     ```

  2. **Dereference operator** (*****)

     – returns the ***value*** of the variable at the address held by the pointer

Tip: The address of a variable of type **`foo`** has type **`foo*`**

& and * Operators

- Declarations and Assignments

C Code Fragment	Comment

`double x = 1.0;`
`double y = 2.0;`
	Create two floating point numbers

`double* z_ptr;`	Create a pointer to a generic double precision floating point number. Note: This VALUE is undefined
`z_ptr = &x;`	Copy the memory address of “x” into z_ptr. z_ptr now “points to x”
`y = *z_ptr;`	“*” de-references z_ptr, which points to “x”, so it returns the value of “x” and stores it into “y”. This is the hard way to say y = x;

Exercise: Draw memory diagram that explains above example

Memory Diagram

Memory Location	Object	Value	Comment
10000x	X	1.0	
10008x	Y	2.0	
10010x	`z_ptr`	???	This is undefined, we have no idea what is in memory here. This is the source of lots bugs! If this pointer WAS set to NULL then we would KNOW it is 0x
10014x			Next free spot. Why 14x and not 18x?
	`z_ptr = &x;`		Set the pointer
10010x	`z_ptr`	10000x	Now z_ptr has a value
	`y = *z_ptr`		The processor will "go to" location 10000x and get the value, then put it in y
10008x	Y	1.0	Value of y changed

`NULL` and `void`

- The **`NULL`** pointer
  - ***"points to nothing"***, *e.g.*, does not reference anything.
  - Often used to avoid "bad pointer values"
  - Use to tell fellow programmers your pointer is "undefined"

- A **`void`** pointer
  - ***"typeless or generic pointer"***,
  - Can be cast to any other type of pointer
  - Use to tell your fellow programmers to use the pointer in their favorite way.

➢ Warning: a **`void`** pointer cannot be dereferenced without casting.

Reading Pointers

- Read pointer declarations from right to left

```
/* a_prt is a pointer to an int */
    int* a_prt;
```

- Warning: When a pointer is declared the compiler allocates memory to hold its value. The declaration does not assign a pointee (address) to the pointer so *the* ***pointer starts with a “bad value”***

Function Pointers

- Functions are just addresses in memory and can also be accessed via pointers.

```
int fun1(int a);

int main(int argc, char *argv[]) {
   /* Declare a function pointer with 1 parameter that returns an int */
   int(*functionPtr)(int);
   int ans;

   functionPtr = &fun1;          /* Initialize the pointer */
    ans = (*functionPtr)(5);   /* Use the function pointer*/
      return(ans);
      } /* main() */
```

  - Notice you can pass values and return values

Practical Pointers

- Confusing pointers and variables is a constant problem in "C"
  - Professionals follow a pointer naming convention like:
    "var_p", "var_ptr", "p_var" or pvar

- E.g.

```
Int count = 1;              /* A variable */
Int count_p = &count;       /* A pointer */
```

- Recommended

Example: **memory_ex01.c**

```c
/* This program will generate warnings and errors, fix them        */
/* Note the value of the address of a and pnum                      */
/* Q: which is in the stack and which is in the heap?               */
#include <stdio.h>  /* for printf */
#include <stdlib.h> /* for malloc */

int main(void){
  int a;              /* integer variable a  */
  int* pnum;          /* pointer to integer  */
  void* pnada=NULL;   /* pointer set to NULL */

  /* Allocated memory for an integer at address pnum */
  pnum = (int *) malloc(sizeof(int));

  /* print addresses */
  printf("Address of a %p\n",&a);
  printf("  value of a %d\n",a);

  printf("Address of  %p\n",pnum);
  printf("pointee of b %d\n",*pnum);

  printf("Address of pnada %p\n",pnada);
  printf("pointee of pnada %d\n",*pnada);
  return 0;
}
```

Compiler Warnings

```
printf("Address of a %p\n",&a);          /* line 20 */
printf("  value of a %d\n",a);

printf("Address of  %p\n",pnum);         /* line 23 */
printf("pointee of b %d\n",*pnum);

printf("Address of pnada %p\n",pnada);  /* line 26 */

printf(“pointee of pnada %d\n",*pnada);
```

```
gcc -ansi -g -Wall -pedantic  memory_ex01.c  -o mem


memory_ex01.c:20: warning: format ‘%p’ expects type ‘void *’, but
argument 2 has type ‘int *’
memory_ex01.c:23: warning: format ‘%p’ expects type ‘void *’, but
argument 2 has type ‘int *’
memory_ex01.c:26: warning: format ‘%p’ expects type ‘void *’, but
argument 2 has type ‘int *’
```

Tip: To print pointers use the conversion string **`%p`** with casting **`(void*)`**
I often use **%xX** instead. (prints the hex pointer value AND prints a “X”

Output

```
int a;                  /* integer variable a  */
int* pnum;              /* pointer to integer  */
void* pnada=NULL;       /* pointer set to NULL */
pnum = (int *) malloc(sizeof(int));
printf("Address of a %p\n",(void *)&a);
printf("  value of a %d\n",a);
printf("Address of  %p\n", (void *)& pnum);
printf("pointee of b %d\n",*pnum);
printf("Address of pnada %p\n", (void *)& pnada);
printf(“pointee of pnada %d\n",*pnada);
```

Output	Comment
`Address of a 0x7fff2b50775c`	Some place high in memory (on the stack)
`value of a 0`	We got lucky here, we never initialized this variable
`Address of pnum 0x1b4f010`	Much lower in memory (in the heap)
`pointee of pnum 0`	Again, we got lucky!
`Address of pnada (nil) Segmentation fault (core dumped)`	Pnada was SET to NULL (0x). We don't have permission to access memory at 0x. The operating system owns this memory and protects it

Casting Pointers

- We can explicitly cast any pointer type to any other pointer type

  Example:

  ```
  float* ptr;
  int* pptr;
  . . .
  pprt = (int*) ptr;
  ```

- Careless use of this “feature” is common cause of ***segmentations faults*** and other (very hard to find) errors.
  - Well designed programs need little casting

Pointer Arithmetic

- Pointer arithmetic is just “address arithmetic”
- Example:

```
sometype* ptr;
. . .
prt + 4;
/* same as */
prt + 4*sizeof(sometype)
```

- Question:
  - ❖ What if `sometype` is `void` ?

    What is the length of something without a length?

Pointer Arithmetic

Q: What is printed ? (answer before running)

```
int main(void) { /* p_arith.c */
  int i;                           /* loop counter */
  char a[4]={'A','B','C','D'};     /* array of characters */
  /* declare pointers */
  char* b_ptr = NULL;              /* char takes 1 byte */
  int*  i_ptr = NULL;              /* int takes 2 bytes */
  /* initialize pointers, e.g., set pointee */
  b_ptr = a;
  i_ptr = (int*)b_ptr;

  /* scan memory in byte increments */
  printf("Scanning by byte increments\n");
  for (i=0;i<4;i++) {
      printf("[%d] Addr:%p, val:%d\n",i,(void*)b_ptr,*b_ptr);
      b_ptr++;
  }

  /* scan memory in word increments */
  printf("Scanning by word increments\n");
  for (i=0;i<4;i++) {
      printf("[%d] Addr:%p, val:%d\n",i,(void*)i_ptr,*i_ptr);
      i_ptr++;
  }
  return 0;
}
```

Pointer Arithmetic

Answer

```
Scanning by byte increments
[0] Addr:0x7fffe9340600, val:A
[1] Addr:0x7fffe9340601, val:B
[2] Addr:0x7fffe9340602, val:C
[3] Addr:0x7fffe9340603, val:D

Scanning by word increments
[0] Addr:0x7fffe9340600, val:1145258561
[1] Addr:0x7fffe9340604, val:56
[2] Addr:0x7fffe9340608, val:4195808
[3] Addr:0x7fffe934060c, val:3
```

Where does 1145258561 come from?

1145258561 (dec) = 0x44434241 0x41="A" 0x42="B"

= "DCBA"

Where does 56 come from?

Pointers in C: Example

Q: What is printed ? (answer before running)

```
/* ex0_pointers.c */
#include <stdio.h>
#include <inttypes.h>   /* c99 uintptr_t */

int main ()
{
   int num = 289;     /* declare and set integer variable */
   int* iptr = NULL;  /* declare pointer to int    */

   iptr = &num;       /* set pointee */

   /* actual size of address depends on machine, e.g. 64bits, 32bits, etc */
   printf("c99: Addr: 0x%lX ,Val:%4d\n",(uintptr_t) iptr, *iptr); /* c99 */
   printf(“Addr: %p ,Val:%4d\n",(void *)iptr, *iptr);             /* portable */

   *iptr = 45;
   printf("Addr: %p ,Val:%4d\n",(void *) iptr, *iptr);

   iptr = NULL;
   printf("NULL pointer Addr: %p\n",(void *) iptr);
fflush(stdout);
   *iptr = 45;
   printf("Addr: %p ,Val:%4d\n",(void *) iptr, *iptr);

   return 0;
}
```

To write portable code that prints memory addresses use **%p**

Pointers in C: Example

```c
int num = 289;
int* iptr = NULL;
iptr = &num;

printf("c99: Addr: 0x%lX ,Val:%4d\n",
        (uintptr_t) iptr, *iptr);
/* portable */
printf("Addr: %p ,Val:%4d\n",
        (void *)iptr, *iptr);

*iptr = 45;
printf("Addr: %p ,Val:%4d\n",
        (void *) iptr, *iptr);

iptr = NULL;
printf("NULL pointer Addr: %p\n",
        (void *) iptr);

*iptr = 45;
printf("Addr: %p ,Val:%4d\n",
        (void *) iptr, *iptr);
```

```
******* Results *******

c99: Addr: 0x7FFF9B34E574,
       Val: 289

Addr: 0x7fff9b34e574 ,Val: 289

Addr: 0x7fff9b34e574 ,Val:  45

NULL pointer Addr: (nil)

Segmentation fault (core
dumped)
```

To write portable code that prints memory addresses use **%p**

Safe Pointers Rules

- A pointer may only be dereferenced after it has been assigned to a pointee.
  Most pointer bugs involve violating this rule.

- Allocating a pointer does not automatically assign it to refer to a pointee. Assigning the pointer to refer to a specific pointee is a separate operation **which is easy to forget**.

- Assignment between two pointers makes them refer to the same pointee.

Reference: Pointers and Memory – Nick Parlante 2000

Pointers in C

- Pointers are at the core of the C programming language.

- They are used for
  - Passing function arguments by reference
  - Dynamic memory allocation
  - Input/Output operations
  - Passing function names as arguments

– Helpful reference: http://boredzo.org/pointers/

Applied Programming

Overview of the C language Part III

Applied Programming

- User-defined Data Types
- Composite Data Types
  - Structures

User Defined Data-Types

- C provides a mechanism to create your own types, e.g., **"typemarks"** (even if that typemark is the same as a primitive data type), *e.g.*, take look at **`stdlib.h`**

- Example: **`<stdint.h>`** (C99)

```
  typedef signed char int8_t;
  typedef short int   int16_t;
  typedef int         int32_t;
```

➢ **`typedef`** creates a typemark (*i.e. alias*); after including **`<stdint.h>`** we can use "**`int8_t, int16_t`** and **`int32_t`**" to declare variables of type "**`signed char, short int`** and **`int`**'' respectively.

Uses of Typemarks in C

- Useful to make programs
  1. ***Readable***
     - **`int32_t`** is more informative than **`int`**
  2. ***Portable***
     - To port this program to another machine we only need to figure out what modifiers are needed to get 32-bit integers and change the typemark, *e.g.*, on a 16-bit machine…
       - **`typedef long int int32_t;`**
  3. ***Maintainable***
     - If we decide that 32-bit integers are not appropriate for the application then we can globally replace "**`int32_t`**" with say "**`int16_t`**"

Use meaningful typemarks

Example

- A few lines of the **`typedefs.h`** header for libraries to develop C programs in a **Freescale microprocessor**

```
. . .
typedef signed char                int8_t;
typedef unsigned char              uint8_t;
typedef volatile signed char       vint8_t;
typedef volatile unsigned char    vuint8_t;
typedef signed short               int16_t;
typedef unsigned short            uint16_t;
. . .
```

Q: what is **`volatile`** ? (more about this later)

Type Casting

- As in Java, we use “type casting” to convert from one type to another (including your own typemarks)
- Example: (I want my “chars” to be unsigned)

```
typedef unsigned char myChar;
int main(void)
{
   int y = 3;
   float x;
   myChar a;
   myChar* a_ptr;
   x = (float) y;
   a = (myChar) y;
   a_ptr = (myChar*)&x;
   return(0);
}
```

Composite Data Types

- Structures
- Unions
- Enumerations
- Arrays

Structures

- Structures:
  - Collection of related variables grouped under a single name (like a Java class but only with data, no methods).
- Defining new types with `struct`

```
struct point {
  double x;
  double y;
};
```

```
struct student {
  char f_name[256];
  char l_name[256];
  int u_id;
};
```

We have created two new datatypes: `point` and `student`

- Initialization and Assignment

```
double x0,y2;
struct point p0;                /* declaration    */
struct point p1={1.0,-3.5};     /* initialization */
struct point p2={2.0,-6.0};
p0 = p1;                     /* assignment */
x0 = p0.x;                   /* accessing members */
y2 = p2.y;                   /* with the . operator */
```

Structures with Pointers

- It is common to have structures with pointer members.

```
typedef struct Darray {
  unsigned int capacity;    /* Max # of elem can hold */
  unsigned int last_used;   /* Index of Last element  */
  int*    payload;         /*  ptr to data           */
} darray;
```

- Then we can use it as follows

```
darray nada; /* declaration */
int mydata[10] = {1,2,3,4,5,6,7,8,9,10};
/* initialize and empty dynamic array */
nada.capacity = 0;
nada.last_used =0;
nada.payload=&mydata;
```

Note: `Darray is the struct name`
`darray is the variable name`

Structures and Pointers

- Example using structure variables and pointers:

```
typedef struct Darray {
  unsigned int capacity;     /* Max # of elem can hold */
  unsigned int last_used;    /* Index of Last element  */
  int*     payload;          /*  ptr to data           */
} darray;
```

- Code fragment:

```
darray  nada;         /* data declaration                */
darray *nada_p        /* pointer declaration             */
nada_p = &nada;       /* pointers MUST point to data     */

nada.capacity    = 100;    /* for data use “.”         */
nada_p->capacity = 200;    /* for pointers use “->”    */
```

Nested Structures

- Structures:
  - Collection of related variables grouped under a single name (like a Java class but only with data, no methods).
- Defining new types with **`struct`**

```
    typedef struct {
      double x;
      double y;
    } point;
```

```
    Typedef struct {
      point A;
      point B;
      point C;
    } triangle;
```

The members of **`triangle`** are **`point`** structures

- Initialization and Assignment

```
  point p1={0.0,0.0};
  point p2={1.1,3.0};
  point p3={-1.0,2.1};
  triangle t={p1,p2,p3};
  printf("A =(%f,%f)",(t.A.x,t.A.y); /* note nested access */
```

Unions

- Unions:
  - Can hold a group of variables of different types (and sizes) in the **same memory location**

Example: A Number

```
union number
{
 float  real;
 unsigned int integer;
};
number.real = 3.1416F;
number.int = 10U;
```

  - The size of a union is the size of the largest member.
- Useful for building/decoding chip register sets

Unions & Structures

- Build an integer using 2 bytes

```
union {
uint16_t   number;
struct {uint8_t high; uint8_t low;} bytes;
} parts;
```

- parts.bytes.high= 1;

  parts.bytes.low = 2;

- parts.number will be 513   (or 258, why?)

Unions & Structures

```
union {uint16_t   number;
          struct {uint8_t high; uint8_t low;} bytes;
          } parts;
```

	number	
	High	Low
Decimal	1	2
Binary	00000001	00000010

- parts.number
  - Big endian   = 0000000100000010 = 258
  - Little endian = 0000001000000001 = 513

Bit fields

- Bit Fields:
  - A bit field is a ***set of adjacent bits*** within a singled `word' (2 bytes)

Example: Floating Point Number Representation

```
struct IeeeSp { /* single precision IEEE float */
 unsigned int fraction:23;
 unsigned int exponent:8;
 unsigned int sign:1;
};
```

- The number after the colon determines the width of the field
- Each variable should be declared as **`unsigned int`**

Enumerations

- Enumerations:
  - Used to create an sequence of integer constants.

Example: (from "C for Java Programmers")

```
/* defines the set ’workdays’:
 * monday = 42, tuesday = 55, wednesday = 56, etc.
 */
enum workdays {monday = 42, tuesday = 55,
               wednesday, thursday, friday };
/* declares variable of the type ’workdays’ */
enum workdays today;
today = tuesday; /* today has value _____ */
today = friday;  /* today has value _____ */
```

Exercise 1

- We are running on a 32 bit machine and given:

```
int x [100];
int y = 100;
int z [y];
```

and memory starts at 10,000 (dec)

Where are x[0], x[1] and y located (in decimal)

- x[0] will be at location 10,000 (dec)
  x[1] will be at location 10,004 (4 bytes = 32 bits).
  y will be at location 10,400 (dec)

Exercise 2

- We are running on a 32 bit machine and given:

```
int x[100];
char *y_p;
int  *z_p;
y_p = (char *)x;
z_p = (int *)x;
y_p++;
z_p++;
```

and memory starts at 10,000 (dec)

What are the values of y_p & z_p?

- y_p & z_p both start at 10,000 dec
- y_p is type char (1 byte) so the final value of y_p is 10,001
- z_p is type int (4 bytes) so the final valueof z_p is 10,004