Data Storage in C

- Pointer to structure
- Storage Classes
- Type Qualifier
- Dynamic Memory Management

Pointer to structure

Expensive to pass large structures between functions.

Instead pass pointer to structure:

```
struct pointer
{
  int x;
  int y;
}
void printpoint (struct point *pp)
{
  printf("(%d, %d)", (*pp).x, (*pp).y);
}
```


Pointer to structure(Continued)

Operator $->$ is shorthand for access a field given by a structure pointer

```
void printpoint (struct point *pp)
{
  printf("(%d, %d)", pp->x, pp->y);
}
```


Pointer to structure(Continued)

Operators $->$ and . associate left-to-right, have high precedence alone with () and []

For example, $++p->x$ increments field x of pp, not the pointer pp

Storage Classes

Every variable and function has a type and a storage class. There are four storage classes.

- auto
- extern
- register
- static

Storage Class *auto*

Variables declared within function bodies are automatic by default, but storage class can be given explicitly:

auto int a, b, c;

Memory allocated upon entering block, released at exit, so values are not kept between invocations.

Remember initialize them before you refer them for the first time

Storage Class *extern*

Variable declared outside functions, and all functions themselves, are external storage class. The storage of extern class is permanently assigned.

The keyword *extern* tells the compiler exist a variable space with associate name either in the same file or in another file.

External variable can be used to transmit information across blocks and functions.

Example of External Variables

```
#include <stdio.h>

char strA[80] = "A string to be used for demonstration";
int f(void); /* function prototype */

int main(void)
{
    char *pA;     /* a pointer to type character */
    char *pB;     /* another pointer to type character */
    puts(strA);   /* show string A */
    ......
}
```

Storage Class *register* (Continued)

When turn on optimization flag -O2, compile gcc will change many access of auto, even global variable to register access.

Because register variable not necessarily stored in memory, it is illegal to take the address of such a variable.

```
register int i;

for (i = 0; i< n; i++)
    ...      /* illegal to refer to &i */
```


Storage Class *static extern*

When applied static to external declarations of functions or variables, scope is restricted to the remainder of current file.

Functions in other files can't access external static variables, even if they attempt use the extern storage class keyword.

Good way to implement information hide in C, like private variable and methods in Java, but limited.

Storage Class *static*

First apply static to variable define within a block.

Comparing with *auto* class, *static* class variables retain their values between invocations.

Variable will be initialized when the block is invoked for the first time.

Storage Class *register*

Advises (but doesn't require) compiler to store value in CPU register rather than in memory.

Purpose is to speed program execution by keeping frequently accessed variables (loop counters) immediately available.

Storage Class *static extern* (Example)

Consider implementation of a stack with operations push(i), pop(), empty() and full()

We will implement with integer array s, using variable next to point to next free element, both variables are declared as static.

```
static int s[MAX_SIZE], next = 0;

void push(int i) { s[next ++] = i;}
int pop(void) { return s[-- next];}
int empty(void) {return next == 0;}
int full(void) {return next == MAX_SIZE;}
```


Type Qualifier *const and volatile*

Type qualifier come after the storage class, if any, but before the type.

Const: the value of this variable can not be change

Volatile: the value of this variable may be modified in some unspecified way.

These qualifiers help the compile optimizing the code.

Dynamic Memory management - functions

```
#include <stdlib.h>

void *calloc(size_t nmemb, size_t size);
void *malloc(size_t size);
void free(void *ptr);
void *realloc(void *ptr, size_t size);
```


Dynamic Memory Management (allocation)

calloc() allocates memory for an array of nmemb elements of size bytes each and returns a pointer to the allocated memory. The memory is set to zero.

malloc() allocates size bytes and returns a pointer to the allocated memory. The memory is not cleared.

malloc() and calloc() will fail if there is not enough available memory.

Dynamic Memory management (strings)

When copying strings, remember that all strings have a terminating 0 character. If you use strlen to count the characters in a string for you, that count will not include the trailing 0, so you must add one before calling malloc:

```
char *somestring, *copy;
 ...
copy = malloc(strlen(somestring) + 1);   /* +1 for \0 */
/* incomplete -- malloc's return value not checked */
strcpy(copy, somestring);
```


Dynamic Memory management (Return)

When malloc is unable to allocate the requested memory, it returns a null pointer. A null pointer, remember, points definitively nowhere. It's a "not a pointer" marker; it's not a pointer you can use.

A call to malloc, with an error check, typically looks something like this:

```
int *ip = malloc(100 * sizeof(int));
if(ip == NULL)
        {
        printf("out of memory\n");
        exit or return
        }
```


Dynamic Memory management (free)

free()

frees the memory space pointed to by ptr, which must have been returned by a previous call to malloc(), calloc() or realloc(). Otherwise, or if free(ptr) has already been called before, undefined behaviour occurs. If ptr is NULL, no operation is performed.

Dynamic Memory management (realloc)

realloc()

changes the size of the memory block pointed to by ptr to size bytes. The contents will be unchanged to the minimum of the old and new sizes; newly allocated memory will be uninitialized. If ptr is NULL, the call is equivalent to malloc(size); if size is equal to zero, the call is equivalent to free(ptr). Unless ptr is NULL, it must have been returned by an earlier call to malloc(), calloc() or realloc().

Dynamic Memory management (Pointer Safety)

Ensuring that the memory they point to is valid.

When a pointer doesn't point where you think it does, if you inadvertently access or modify the memory it points to, you can damage other parts of your program, or (in some cases) other programs or the operating system itself!

One particular problem concerns functions that return pointers. Where is the memory to which the returned pointer points? Is it still around by the time the function returns? Dynamic allocation gives us a lot more flexibility and responsibility that we manage dynamically allocated memory carefully.

Dynamic Memory management (Misc)

In big project, when define function interface and using call by reference, it's a good idea to specify who will deal with which part of memory management issue.

```
char *strcpy(char *dst, char *src);
char *strdup(const char *s1);
char *strstr(const char *s1, const char *s2);
```