Pointers

• A pointer is a memory address

• Pointer variables are variables whose values are memory addresses

• Referencing a value though a pointer is called indirection

• Pointer variable declaration in C:
  int * countPtr, count;

  – countPtr is a variable of type “int *”, i.e., a pointer to an int
  – count is a variable of type int.

Use of Pointers in C

Pointers are a low-level mechanism that allows to simulate important high-level constructs:

• call by reference

• (certain) subarrays

• recursive datastructures

• graph-like datastructures

• higher-order functions

But: Pointers must be used very carefully!

The Address Operator &

int y = 5;
int * yPtr;

yPtr = &y;

• The operand of & must be a variable

• & can not be applied to
  – constants
  – expressions
  – register variables

• Type rule: If e has type t, then &e has type t *.

The Pointer Dereferencing Operator *

int y = 5;
int * yPtr;

yPtr = &y;

printf( "%d\n", *yPtr );
*yPtr = 7;
printf( "%d\n", y );

The dereferencing operator, applied to a pointer value p, returns the variable p points to, i.e.,

• in an expression: the contents of the memory address p
• to the left of an assignment: the variable at memory address p

Make sure p points to a sensible address!

Type rule: If e has type t *, then *e has type t
Simulate Execution! — 1

```c
#include <stdio.h>

void swap(int * p, int * q)
{
    int h = *p;
    *p = *q;
    *q = h;
}

int main ()
{
    int i=4, j=7;
    swap( &i, &j );
    printf("i = %d; j = %d\n", i, j);
    return 0;
}
```

Simulate Execution! — 2

```c
#include <stdio.h>

void swap(int * p, int * q)
{
    int h = *p;
    *p = *q;
    *q = h;
}

int main ()
{
    int i=4, j=7;
    swap( &i, &j );
    printf("i = %d; j = %d\n", i, j);
    return 0;
}
```

Simulate Execution! — 3

```c
#include <stdio.h>

void swap(int * p, int * q)
{
    *p += *q;
    *q -= *p;
    *p += *q;
    *q = 0 - *q;
}

int main ()
{
    int i=4, j=7;
    swap( &i, &j );
    printf("i = %d; j = %d\n", i, j);
    return 0;
}
```

Relationship between Pointers and Arrays

Arrays are like
- `const` pointers
- for which the size of space pointed to is known

Assume:
- `double b[5];`
- `double * bPtr;`
- `bPtr = b;`

Then:
- `bPtr == &b[ 0 ]`
- `bPtr + 1 == &b[ 1 ]`
- `*(bPtr + 3) == b[ 3 ]`
- `bPtr + 3 == b + 3`
- `bPtr[ 1 ] == b[ 1 ]`
Arrays of Strings

#include <stdio.h>

int main() {
    int i;
    for (i = 0; i < 4; i++)
        printf("suit[%d] = %p	: "%s"
", i, suit[i], suit[i]);
suit[3] = "Shovels";
    for (i = 0; i < 4; i++)
        printf("suit[%d] = %p	: "%s"
", i, suit[i], suit[i]);
    return 0;
}

• char * suit[4] — an array of char * values
• char * values interpreted as beginning of zero-terminated character strings
• Can be considered as an array of arrays — suit[2][3] works.
• Subarrays can be anywhere — nothing is known about relative arrangement in memory of suit[2] and suit[3]

Command-Line Arguments

#include <stdio.h> // argvS.c

int main(int argc, char *argv[]) { char * argp;
    argp = argv;
    while ( *argp != NULL ) {
        printf("argv[%d] = %p	: "%s"
", argp - argv, *argp);
        argp++;
    } return 0;
}

• The array char *argv[] contains argc+1 elements.
• argv[argc] == NULL
• Some people declare their main functions with int main(int argc, char **argv)
• This declares the intent to use argv as the beginning of a NULL-terminated “string of pointers”

#include <unistd.h> // flip.c

int main(int argc, char *argv[]) {
    char * newargv[argc];
    int i;
    for (i=1; i < argc; i++) newargv[i-1] = argv[i]; // copying argv
    if (argc >= 4) {
        newargv[1] = argv[3];
        newargv[2] = argv[2];
        return execvp(argv[1], newargv);
    } else return 1;
}

• The command line consists of space-separated words:
  – the command, and
  – arguments
• argv contains the whole command line
• char * argv[] can be used as an array with argc elements:
  – argv[0] is the command, and
  – argv[1] ... argv[argc-1] are the arguments
  – the number of arguments is (argc-1)!
• char * argv can also be used as the beginning of a NULL-terminated “string of pointers”
const

const variables cannot be assigned to — but still have addresses:

```c
#include <stdio.h>
int main ()
{
    const int n = 42;
    printf("%d %p\n", n, &n);
    return 0;
}
```

The `sizeof` Operator

`sizeof` is a keyword used like a function that accepts as single argument
- any variable, or
- any type.

and returns an integral value of type `size_t` indicating
- how many bytes are reserved for the given variable, or
- how many bytes are reserved for variables of the given type.

Note: For array variables this yields `sizeof(element_type) * array_size`.

General rules:
- a byte has 8 bits
- in C, characters are 8-bit integral values
- on a `n`-bit architecture, `int` and pointers occupy `n` bits, i.e., `n/8` bytes
- (double variables occupy twice as much space as floats)
- `long` occupies not less space than `int`

Const and Pointers

- **constant**: read-only
- **non-constant**: variable, read-write

```c
int * p;  // `p` contains a non-constant pointer to non-constant data
const int * p;  // `p` contains a non-constant pointer to constant data
int * const p;  // `p` contains a constant pointer to non-constant data
const int * const p;  // `p` contains a constant pointer to constant data
```

Pointer Expressions and Pointer Arithmetic

For pointer arithmetic, a `T`-pointer `ptr` should be understood as an

"abstract index into memory considered as an array of T-variables"

Therefore:
- `ptr + 1` is "the next index" — it points to the next `T`-variable
- when considering pointers as integers (for example when printing with "%p")
  the difference between `ptr + 1` and `ptr` is: `sizeof(T)`
- for pointers of the same type (e.g. after `ptrB = ptr + n`)
  one may calculate the pointer difference `ptrB - ptr`, which will be `n`

Other pointer arithmetic operators:
- `*`, `*=` , `++`, `--`
void Pointers

void * ptr;

- *ptr* is a void pointer — a “raw address”
- any pointer value can be assigned to *ptr*
- *ptr* can be assigned to any pointer variable
- void pointers are used as “pointers to anything” — faking polymorphism
- void pointers cannot be dereferenced — a cast is necessary first

```c
#include <stdio.h>

int main() {
    char s[] = "Hello World!";
    void * p = s;
    void * q = p + 1;
    printf("%d "%s"\n", sizeof(void), (char *)q);
    return 0;
}
```

### NULL

- NULL is defined in *stdio.h* as the zero-value for pointers
- NULL must not be dereferenced
- NULL is the only pointer value for which you can determine in a safe way that you are not allowed to dereference it
- The presence of NULL allows pointers to be used as optional references:
  - Each pointer value — either is a reference to a variable
  - or is NULL

```c
void myInit(int * p) { /* p is either a reference or NULL */
    if (p ≠ NULL) {
        p = getLogLines(logfile);
        log("Initialisation message: New run\n");
    }
    else
        unlink(logfile);
}
```

### Stack vs. Heap

- Local variables, function arguments, return values, and return addresses are kept in stack frames on the execution stack
- The stack “grows” and “shrinks” with the number of nested function calls.
- Consecutive function calls use the same stack space.
- Therefore, if a “new variable” needs to be accessible after a function returns, it cannot be allocated on the stack.
- The heap is the space for dynamic data:
  - void *malloc(size_t size) allocates size bytes on the heap and returns a pointer to the allocated memory (from *stdlib.h*).
  - void free(void *ptr) frees the memory space pointed to by *ptr*, which must have been returned by a previous call to malloc().

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

char * reverse(int length, char * string) {
    char result[length+1];
    int i, j;
    for (i = length-1, j=0; i ≥ 0; i--, j++) result[j] = string[i];
    result[length] = '\0';
    return result;
}
```

```c
int main() {
    char msg1[] = "Hello world!";
    char * msg2;
    msg2 = reverse(strlen(msg1), msg1);
    printf("Reversing finished!
");
    printf("msg2='%s'
", msg2);
    return 0;
}
```
 include <string.h>

char * strdup(const char *s);

The *strdup*() function returns a pointer to a new string which is a duplicate of the string *s*. Memory for the new string is obtained with malloc(3), and can be freed with free(3).

The *strdup*() function returns a pointer to the duplicated string, or *NULL* if insufficient memory was available.

---

### strdup

#### Parameterised Sorting Using Function Pointers

```c
int leq(double x, double y) { return x ≤ y; }
int geq(double x, double y) { return x ≥ y; }

void sort(double a[], const int size, int (*compare)(double x, double y)) {
    ...
    if ( (*compare)(a[i], a[i+1]) ) {
        ...
    } else {
        ...
    }
    ...

int main() {
    ...
    sort(b, SIZE, leq);
    ...
}
```

---

### Points to Functions

```c
int leq(double x, double y) { return x ≤ y; }
int geq(double x, double y) { return x ≥ y; }

This defines two functions:

- at runtime, functions are machine code fragments, stored at some address
- therefore, at run-time, the name *leq* is bound to an address
- *leq* is a **function pointer value**
- *leq* can be passed as argument to functions, or assigned to pointer variables
- The type of a variable or argument *compare* accepting binding to *leq* is:
- int (*compare)(double x, double y)

- Full prototype:
  ```c
  void sort(double a[], const int size, int (*compare)(double x, double y));
  ```
- Short prototype:
  ```c
  void sort(double[], const int, int (*)(double x, double y));
  ```
- Use of function pointer:
  ```c
  if ( (*compare)(a[i], a[i+1]) ) ...
  ```
- Invocation:
  ```c
  sort(b, SIZE, leq);
  ```

---

### Exercise: count_maximum

**Design** and implement a C function *count_maximum* that, given an int array,

- finds the maximum element in this array
- with respect to an ordering passed to *count_maximum* as a **function pointer argument**,
- and also counts how many times this maximum occurs,
- and makes both the maximum and this count available to its caller.
- The array should be **traversed only once**!
#include <stdio.h>

char a[] = "Hello world!";
char * p = a;
char * s = "Hello world!";

int main () {
    a[5] = '~';
    printf("p = %s\n", p);
    p[5] = '_';
    printf("p = %s\n", p);
    printf("s = %s\n", s);
    s[5] = '_';
    printf("s = %s\n", s);
    return 0;
}

The Character Handling Library (ctype.h)

```c
int isdigit(int c); /* checks for a digit (0 through 9) */
int isalpha(int c); /* checks for an alphabetic character */
int isalnum(int c); /* checks for an alphanumeric character; equivalent to (isalpha(c) || isdigit(c)) */
int isxdigit(int c); /* checks for a hexadecimal digits, i.e. one of 0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F */
int islower(int c); /* checks for a lower-case character */
int isupper(int c); /* checks for an uppercase letter */
int topper(int c); /* converts the letter c to upper case, if possible */
int tolower(int c); /* converts the letter c to lower case, if possible */
int isspace(int c); /* checks for white-space characters. */
    In the C and POSIX locales, these are: " \f \n \r \t \v" */
int iscntrl(int c); /* checks for a control character */
int ispunct(int c); /* checks for any printable character which is not a space or an alphanumeric character */
int isprint(int c); /* checks for any printable character including space */
int isgraph(int c); /* checks for any printable character except space */
```

String Conversion Functions from stdlib.h

No error detection:
```
int atoi(const char *nptr);
long atol(const char *nptr);
long long atoll(const char *nptr);
double atof(const char *nptr);
```

Setting endptr to first invalid character, and setting errno:
```
long int strtol(const char *nptr, char **endptr, int base);
long long int strtoll(const char *nptr, char **endptr, int base);
unsigned long int strtoul(const char *nptr, char **endptr, int base);
unsigned long long int strtoull(const char *nptr, char **endptr, int base);
```
```
double strtod(const char *nptr, char **endptr);
float strtof(const char *nptr, char **endptr);
long double strtold(const char *nptr, char **endptr);
```
Using `strtod`

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

int main(int argc, char * argv[]) {
    double d;  /* variable to hold converted number */
    char * restPtr;  /* pointer variable to hold rest pointer */

    errno = 0;
    d = strtod(argv[1], &restPtr);

    if (errno) perror("ERROR in strtod");
    printf("Conversion of the string \"%s\" produces the double value \"%.2g\", d);
    printf(" and leaves the remainder string \"%s\", restPtr);

    return 0;
}
```

String Manipulation Functions (string.h)

```c
size_t strlen(const char *s);

char * strcpy(char *dest, const char *src);
char * strncpy(char *dest, const char *src, size_t n);

char * strcat(char *dest, const char *src);
char * strncat(char *dest, const char *src, size_t n);

int strcmp(const char *s1, const char *s2);
int strncmp(const char *s1, const char *s2, size_t n);

int strcasecmp(const char *s1, const char *s2);
int strncasecmp(const char *s1, const char *s2, size_t n);

int strcoll(const char *s1, const char *s2); /* uses locale */
```

Standard Input/Output (stdio.h)

```c
int getchar(void); /* returns character or EOF */
char * gets(char *s); /* reads up to newline or EOF */

int putchar(int c); /* returns c, or EOF on error */
int puts(const char *s); /* returns EOF on error */

int sprintf(char * str, const char * format, ...);
int snprintf(char * str, size_t size, const char * format, ...);
```

String Search Functions (string.h)

```c
char * strchr(const char * s, int c);
char * strchr(char * s, const char * c);

size_t strspn(const char * s, const char * accept);
size_t strcspn(const char * s, const char * reject);

char * strpbrk(const char * s, const char * accept);
char * strpbrk(char * s, const char * accept);

char * strstr(const char * haystack, const char * needle);
char * strstr(char * s, const char *delim);
char * strstr_r(char * s, const char *delim, char **ptrptr); /* Don't use! */
```

From the man page: The function `snprintf` does not write more than `size` bytes (including the trailing '0'). If the output was truncated due to this limit then the return value is the number of characters (not including the trailing '0') which would have been written to the final string if enough space had been available.
Memory Functions (string.h)

void *memcpy(void *dest, const void *src, size_t n);
   /* ranges must not overlap */

void *memmove(void *dest, const void *src, size_t n);
   /* handles overlapping ranges */

int memcmp(const void *s1, const void *s2, size_t n);

void *memchr(const void *s, int c, size_t n);
void *memrchr(const void *s, int c, size_t n); /* GNU extension */

void *memset(void *s, int c, size_t n);