2.7 Example: Compute string length using a pointer

```
int stringlen(char *s)
{
    int n;

    for (n = 0; *s != '\0'; s++)
        n++;

    return n;
}
```

- Since s is a pointer, incrementing it is perfectly legal.
- s++ has no effect on the character string in the caller function, it merely increments strlen's private copy of the pointer.

Legal calls to `strlen`?

Given `int stringlen(char *s)` as above, which calls will work?

```c
int main(void)
{
        char array[] = "hello";

        /* ok */
        printf("%d\n", stringlen(array));

        /* warning we do not yet understand */
        printf("%d\n", stringlen("hello"));

        return 0;
}
```

- We need to elaborate on this! (*cf.* page 86)
- For now, only pass variables declared as `char[]` to `stringlen`.

Comparison of pointers

Pointers may be **compared** under certain circumstances:

- If `p` and `q` point to members of **the same** array, then comparison like `==`, `!=`, `<`, `>=`, *etc.* work properly.
  E.g., `p < q` is true, if `p` points to an earlier member of the array than `q` does.
- The behaviour is **undefined** for arithmetic or comparisons with pointers that do not point to members of the same array.
- There is one exception: The address of the first element past the end of an array can be used in pointer arithmetic.

Pointer subtraction

- If `p` and `q` point to elements of the same array and `p < q`, then `q-p+1` is the number of elements from `p` to `q` inclusive.

```
#include <stddef.h>    /* includes type ptrdiff_t */

ptrdiff_t stringlen(char *s)
{
        char *p = s;

        while (*p)  /* i.e., *p != '\0' */
                p++;

        return p - s;
}
```

- `p` is initialized to `s`, *i.e.*, point to the first character of the string.
- `while` loop: examine each char until `'\0'` is seen.
- Use pointer subtraction to determine string length.

Valid Pointer Operations

Legal pointer operations summarized

- Assignment of pointers of the same type, or `void *`.
- Assigning or comparing to `NULL`.
- Adding or subtracting a pointer and an integer.
- Subtracting or comparing pointers to members of the **same array**.

Illegal pointer operations

- Multiply, divide, shift, or mask pointers.
- Add `float` or `double` to pointers.
- Assign a pointer of one type to a pointer of another type without cast (exception is `void *`).
- Subtracting or comparing pointers to members of **different arrays**, or not pointing to arrays at all.

2.8 The `const` type qualifier

The **keyword** `const` can be used to make a variable **readonly**.

```
const type var;
```

```
type const var;
```

- Both forms above are **equivalent**.
- General rules:
  - If `const` is **next to a type specifier** (*e.g.*, `int`, `double`, ...), it applies to that type specifier.
  - Otherwise, it applies to the pointer **asterisk to its left**.

Note The position of `const` relative to an `*` is relevant:

A pointer to a **constant object**.

```
type const * var;
const type * var;
```

You may assign to the pointer, but not to its target.

A **constant pointer** to an object.

```
type * const var;
```

You may assign to the target, but not to the pointer.

- Hint: Read pointer declarations from **right to left**.

Examples

```
int i;
int const c = 32, d;
int * const p1 = &i, * p2 = &i, * const p3;
int const * p4;
```

- c and d are constant ints.

```
i = c;   /* ok: copy value from c, and store in i */
c = i;   /* error: assignment of read-only variable c */
d = 23;  /* error: assignment of read-only variable d */
```

- p1, p3 are constant pointers to int, **but** p2 is a pointer to int.

```
p1 = &i;   /* error: assignment of read-only variable p1 */
*p1 = 12;  /* ok: write to the integer, not the pointer! */
p2 = &i;   /* ok: p2 is not const */
```

- p4 is a pointer to a constant int.

```
p4 = &i;   /* ok: p4 is not const */
*p4 = 34;  /* error: assignment of read-only location *p4 */
i = 99;    /* ok: i is not constant */
```

A function can **promise not to modify** a value passed by reference:

```c
1  #include <stdio.h>
2
3  int nice(int const * x)
4  {
5      /* *x = 3; */  /* causes error */
6      return *x + 2;
7  }
8
9  int sloppy(int * x)
10 {
11     return *x + 2;
12 }
```

```c
13 int main(void)
14 {
15     int i = 12;
16     int const j = 23;
17
18     nice(&i);
19     nice(&j);
20     sloppy(&j);  /* causes warning */
21
22     printf("%d\n", i);
23     return 0;
24 }
```

- Passing a reference to a constant object to a function that does not promise not to modify it, causes a **warning**! (line 20)
- A **string literal** in C is constant, and must not be written to!

```c
int stringlen(char * foo);
stringlen("hello");  /* warning */
```

```c
int stringlen(char const * foo);
stringlen("hello");  /* fine */
```

Cast away `const` — pun intended

- Review the warning issued by line 20 on the previous slide:

```
const2.c:28:2: warning: passing argument 1 of ‘sloppy’ discards ‘const’
qualifier from pointer target type [enabled by default]
  sloppy(&j);
```

- If you
  - absolutely must use that function (it may come from a library),
  - and you absolutely know that it will not change the value
  - and you absolutely cannot create a copy and pass that instead,

  then you may cast the type into a non-`const` one:

```c
int sloppy(int * x)
{
    return *x + 2;
}
int modify(int * x)
{
    (*x)++;
    return *x+2;
}
```

```c
int main(void)
{
    int const j = 23;

    sloppy((int*)&j);  /* no warning */
    modify((int*)&j);  /* you're on your own */
    printf("%d\n", j);

    return 0;
}
```

Cast away `const` — broken promise

- A function may break its promise:

```
int evil(int const * x)
{
    *(int *)x = 666;
    return *x + 2;
}
```

```
int main(void)
{
    int const j = 23;
    evil(&j);
    printf("%d\n", j);
    return 0;
}
```

- Writing such functions is a very bad idea:
  - You **break the promise** given in the function's signature!

C strings again

A string constant or string literal, *e.g.*, `"I am a string"`...

- ...is an array of characters, (automatically) terminated with `'\0'`.
- ...occupies one more byte in storage than the number of characters between the double quotes
- Quite often, string constants appear as arguments to functions, *e.g.*

```
printf("%s", "Hello World!\n");
```

  For this to work, the function must have a `const char *` parameter!
- Access to the constants is provided through character pointers, *i.e.*,a string constant is accessed by a **pointer to its first element**.

Note There is no string-copying going on here. Why?

```
char *pmessage;
pmessage = "now is the time";
pmessage = "hello, world";
```

String literals are constant

In C, a **string literal** is a constant, that you **must not write** to.

- Why? May be shared. Stored in a read-only location (*cf.* later).

Examples

- You must not write to literals.

```
char *s1 = "hello";         /* warning: initialization discards const */
s1[3] = 'X';                /* this will segfault (i.e., access violation) */
```

- You cannot pass literals to functions accepting a non-const.

```
const char *s2 = "hello";   /* correct */
int stringlen(char *s);     /* assume we have that function */
stringlen(s2);              /* warning: discards 'const' qualifier */
stringlen("hello");         /* warning, because the literal is const */
```

- Use `const` to indicate where your functions behave nice.

```
int stringlen2(const char *s);  /* assume we have that function */
stringlen2(s2);                 /* correct */
stringlen2("world");            /* correct */
```

Character pointers & character arrays differ

A `char` **array** initialised from a constant is **writable**!

```
const char *s1 = "hello";     /* from previous slide */

char s3[] = "world";   /* correct: writable array initialized from constant */
stringlen(s3);         /* correct */
stringlen2(s3);        /* correct */
s3[1] = 'X';           /* correct: the array is writable */
s3 = s1;               /* wrong: array name used as l-value */
```

- The array is initialized from a literal!
- The array is writable, the literal is not.

How can we copy strings?

```
char t[100];  /* target array */
const char *s = "hello world";

int main(void)
{
    stringcpy(t, s);                        /* we are looking for this */
    printf("%s\n", t);

    return 0;
}
```

Function to copy string `s` into array `t`:

```
void stringcpy(char *t, char const *s)
{
    int i;

    for (i = 0; s[i] != '\0'; i++)
        t[i] = s[i];
    t[i] = '\0';
}
```

Question Why is the `const` *necessary* in the specification of parameter `s`?

String copy using pointers

```
void stringcpy(char *t, char const *s)
{
    while (*s != '\0')
        *t++ = *s++;
    *t = '\0';
}
```

- The value of `*s++` is the character that `s` pointed to before `s` is incremented. (*cf.* page 65)
- The postfix `++` doesn't change `s` until after this character has been fetched.

An even leaner version:

```
void stringcpy(char *t, char const *s)
{
    while ((*t++ = *s++))
        ;
}
```

Standard idioms For pushing and popping a stack

```
*p++ = val;   /* push val onto stack */
val = *--p;   /* pop top of stack into val */
```

Question Using these idioms, what exactly does `p` point to?

3
Dynamic memory management

Current situation Until now, we cannot change the amount of space available to store data:

- The **number of variables** in a C program is fixed in the source code.
- **Arrays** cannot grow, nor shrink.

$\Rightarrow$ Use **excessively large** arrays that are guaranteed to be big enough.
That's not nice!

Dynamic memory Get more memory **on demand**, and only if required.

- First figure out how much memory is needed, then request that from the OS (*aka.* **allocating**).
- Or guess how much is needed and allocate that. Adapt as necessary.
- **Return** unused memory to the OS.

3.1 Allocating memory

malloc(3) and calloc(3) **allocate** blocks of memory.

```
#include <stdlib.h>
void *malloc(size_t size);
void *calloc(size_t num, size_t size);
```

- size_t, defined in stddef.h is an unsigned integral type.
- malloc allocates a block of size bytes of memory.
  - The memory is **not initialised**.
  - Initialisation can be done using memset(3).
- calloc allocates memory for an array of num elements of size bytes each.
  - The storage is **initialised to zero**.
- Both fuctions return a pointer to the (start of) the allocated memory, or NULL if the request cannot be satisfied (or the requested size is 0).
- void * is the proper type for a **generic pointer**.

```
int *ip;
ip = calloc(42, sizeof(int)); /* space for 42 ints */
```

Extend or reduce allocated memory

realloc(3) "modifies" the size of a block of memory previously allocated with malloc(3).

```
#include <stdlib.h>
void *realloc(void *ptr, size_t size);
```

- Changes the size of the object pointed to by ptr to size bytes.
  - Note that it may be necessary to **move** all data to a new location!
- realloc returns a **new pointer** to the (possibly moved) object.
  - Do not use the **old pointer**, it is invalid!
- The contents will be **unchanged** in the range from the start of the region up to the minimum of the old and new sizes.
  - Freshly allocated memory is **not initialised**.
- **Note:** ptr must point to memory previously allocated with malloc, *i.e.*, this will not work:

```
int arr[23];
int *p = arr;
p = realloc(p, 42 * sizeof(int)); /* wrong */
```

3.2 Freeing allocated memory

`free`(3) frees memory previously allocated with `malloc`(3).

```
void free(void *ptr);
```

- If `ptr` is a `NULL` pointer, no action occurs.
- It is an error to dereference something **after it has been freed**.
- Only areas of free memory can be used by `malloc`(3)!
- It is important to free memory you do not need anymore.
  - In general, this is not an easy task.
  - There is **no garbage collector**.
  - If you do not `free`, you may **run out of memory**.
- **Note:** `ptr` must point to memory previously allocated with `malloc`, *i.e.*, this will not work:

```
int arr[23];
free(arr);  /* wrong */
```

3.3 Example

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

int main(void)
{
    int *p = malloc(8 * sizeof(int));  /* allocate mem for 8 int */

    for (int i = 0; i < 8; i++)                 /* write some data */
        p[i] = i*i;

    p = realloc(p, 16 * sizeof(int));  /* get more space */
    p[15] = 100;

    p = realloc(p, 12 * sizeof(int));  /* free some memory */
    /* p[15] = 7; */      /* invalid */

    for (int i = 0; i < 12; i++)                /* print whole memory block */
        printf("%2d\t%d\n", i, p[i]);           /* slots 8–11 contain garbage */

    free(p);  /* free all memory used by p */

    return 0;
}
```

Caution

- **Always free allocated memory** when it's no longer used.
  Only exception: Your program terminates.
- It is a bug not to **check the return values** of `malloc`(3), `calloc`(3), or `realloc`(3) for error conditions.
  Review the example on slide 102!
- One **must not access unallocated memory**, or memory after calling free on it.

Ignoring any of these rules **is a bug** that may, or may not, show up during testing. Even if the program behaves as expected, it is still buggy!

3.4 Handling strings

- With `#include <string.h>` you'll get access to a plethora of string handling functions, documented in string.h(0).
- Example: Copy string pointed to by src, to buffer pointed to by dest.

```
char *strcpy(char *dest, const char *src);                /* cf. strcpy(3) */
```

Question How can we make a copy of a string?

```
const char *msg = "hello world\n";

char *copy;
strcpy(copy, msg);
```

- What do you think about this approach?

```c
const char *msg = "hello world\n";

char *copy;  /* not initialized, points nowhere */
strcpy(copy, msg);
```

Bad idea: The target pointer does not point to any allocated memory!
⇒ **Undefined behavior**[22]

[22] http://blog.regehr.org/archives/213, http://blog.regehr.org/archives/970

Question String copy: What about this one?

```
char *strcpy(char *dest, const char *src);
const char *msg = "hello world\n";

char *copy = malloc(strlen(msg));
strcpy(copy, msg);
```

```
char *strcpy(char *dest, const char *src);
const char *msg = "hello world\n";

char *copy = malloc(strlen(msg)); /* not enough */
/* return value unchecked */
strcpy(copy, msg);
```

- Unchecked if we got any memory at all.
- Even then, not enough memory is allocated: `strlen` returns length *excluding* `NUL`, but `strcpy` copies that as well!

⇒ **Undefined behavior**

Easy to fix:

```
#include <err.h>

char *copy = malloc(strlen(msg) + 1);
if (!copy)
    err(1, "copy");    /* cf. err(3). Terminates with a message like */
                       /* a.out: copy: Cannot allocate memory */
```

Question String copy: Not correct. Why?

```
const char *msg = "Old MacDonald Had a Farm";

size_t len = strlen(msg) + 1;
char *cp1 = malloc(len),
     *cp2 = malloc(len);

if (!cp1 || !cp2)
     err(1, "cp1 or cp2");

for (size_t i = 0; i < 13; i++)  /* copy only first two words */
    cp1[i] = msg[i];

strcpy(cp2, cp1);  /* copy that to cp2 */
```

```
strcpy(cp2, cp1);  /* copy cp1 to cp2 */
```

- `strcpy` will copy bytes from `cp1` until the string ends, *i.e.*, until it sees a `'\0'` character.
- The source `cp1` may not be terminated by a `NUL` character!

⇒ `strcpy` may "fall over the edge", and overwrite adjacent memory!
⇒ **Undefined behavior**

Solution to all these cases:

- Use `strncpy(3)` instead, which will not write more than `n` bytes!

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

  Always be aware of the amount of data to be written!

- *Overflowing fixed-length string buffers is a favorite cracker technique for taking complete control of the machine.* `strcpy`(3)

Note that `strncpy` may **not write** the terminating `NUL`!

4
More on Types

Structures, unions, enumerations
Defining types
Unscrambling C declarations

4.1 Type Conversions

- Type ranking
  - `_Bool` → `char` → `short` → `int` → `long` → `long long` →
  - `float` → `double` → `long double`
- Typing constants
  - `L` (`long`), `LL` (`long long`)
  - `U` (`unsigned`), `UL` (`unsigned long`), `ULL` (`unsigned long long`)
  - `F` (`float`), `L` (`long double`)
- Automatic promotion to (`unsigned`) `int`
  - The signedness of the higher-ranked type takes precedence
  - For equal ranks, `unsigned` takes precedence

4.2 Structures

Declaring structures

- A structure allows a group of variables to be accessed via one name.
- To this end, a structure introduces a **new type**.

The definition has four parts:

```
struct tag {
    /* list of member declarations */
    type name...;
    type name...;
} variable...;
```

- the keyword `struct`
- an optional *structure tag*
- brace-enclosed list of declarations for the **members**
- list of variables of the new structure type (optional)

Declaration examples:

```
struct point {
    double x, y;
};
/* now "struct tag" serves as type name */
struct point p, q;
```

or equivalent

```
struct {
    double x, y;
} p, q;  /* directly name variables */
/* But you cannot reuse this struct! */
```

Using structures

- A list of constant member values in the right order initialises a structure. Or use individual members by name, in any order.

```
struct point p1 = { 320, 200 };
```

```
struct point p2 = { .x = 320 };
```

- Structures can be assigned as a unit, or be returned from a function.

```
struct point p = q;        /* copy all members */
struct point mkpoint();    /* declares function returning a point structure */
```

- Members can be accessed using `name.member`

```
struct point center;
printf("%f, %f\n", center.x, center.y);
```

- There is a shortcut for handling pointers to structs: `ptr->name`

```
struct point origin, *pp;
pp = &origin;                  /* so you can get the address of a structure */
printf("origin is (%f,%f)\n", (*pp).x, (*pp).y);
printf("origin is (%f,%f)\n", pp->x, pp->y);   /* this is equivalent */
```

- Structures can contain other structures

```
struct rect {
    struct point ul;
    struct point lr;
} square;

square.ul.x = 0; square.ul.y = 1;
square.lr.x = 2; square.lr.y = 0;
```

- Structures can be self-referential **via pointers**.

```
struct tnode {              /* the tree node: */
    int value;              /* node label */
    struct tnode *left;     /* left child */
    struct tnode *right;    /* right child */
};
```

- The **size** of a struct may be *larger* than the sum of its members!

```
struct demo {
    int i;
    char c;
};
```

```
/* prints 8 on my machine */
printf("%zu\n", sizeof(struct demo));
```

- Structures can be array elements

```
struct point {
    int x;
    int y;
} points[] = {
    { 0, 1 },
    { 2, 3 },
    { 3, 5 }
};
```

- Structures are passed to functions **by value**!

```
struct point add(struct point p1, struct point p2)
{
    p1.x += p2.x;
    p1.y += p2.y;

    return p1;
}
```

  - The **whole struct** is copied!
  - This also works for the `return` value!

4.3 Unions

- A *union* is a **variable** that may hold (at different times) objects of **different types** and sizes.
- Unions provide a way to manipulate different kinds of data in a **single area of storage**.

The syntax is similar to structures:

```
union tag {
    /* list of member declarations */
    type name…;
    type name…;
} variable…;
```

- the keyword `union`
- an optional *union tag*
- brace-enclosed list of declarations for the **members**
- list of variables of the new union type (optional)

- Union variables will be large enough to hold the **largest** of the member types. (the specific size is implementation-dependent)
- It is the programmer's responsibility to keep track of which member currently holds a value. **Only one** can be used at any time.

```c
union demo {
    int i;
    double d;
    char c;
};

union demo u;

printf("size: %zu\n", sizeof(u));

u.i = 23;   /* now u.d and u.c contain garbage! */
printf("u.i: %-16d   u.d: %-16e   u.c: '%c'\n", u.i, u.d, u.c);

u.d = 4.2;   /* now u.i and u.c contain garbage! */
printf("u.i: %-16d   u.d: %-16e   u.c: '%c'\n", u.i, u.d, u.c);

u.c = 'X';   /* now u.i and u.d contain garbage! */
printf("u.i: %-16d   u.d: %-16e   u.c: '%c'\n", u.i, u.d, u.c);
```

```
$ ./a.out
size: 8
u.i: 23                u.d: 6.952931e-310       u.c: ''
u.i: -858993459        u.d: 4.200000e+00        u.c: '□'
u.i: -858993576        u.d: 4.200000e+00        u.c: 'X'
```

Use case 1: Saving space

- Usually occur as a part of a larger struct that also has implicit or explicit information about the data.
- Used to save space.

Example Zoological information on certain species. First attempt:

```
struct creature {
    char has_backbone;
    char has_fur;
    short num_of_legs_in_excess_of_4;
};
```

However…

- All creatures are either vertebrate or invertebrate.
- Only vertebrates have fur and only invertebrates have more than four legs.
- Nothing has more than four legs and fur.

That is why…

```c
union secondary_characteristics {
    char has_fur;
    short num_of_legs_in_excess_of_4;
};
struct creature {
    char has_backbone;  /* indicates valid union field! */
    union secondary_characteristics form;
};

struct creature naked_mole_rat = {
    .has_backbone = 'y',
    .form.has_fur = 'n'     /* Note the .form prefix */
};
```

Use case 2: Data interpretation

```c
union bits32_tag {
    int whole; /* one 32-bit value */
    char byte[4]; /* four 8-bit bytes */
} value;
```

- Take the whole with `value.whole`
- Take 3rd byte with `value.byte[2]`

Notes

- You need to check your compiler's documentation to make proper use of this!
- Generally, structs are about one hundred times more common than unions.

4.4 Enumerations

- Enumerations provide a convenient way to associate **constant integer** values with **names**.
- An alternative to `#define` with the advantage that the values can be generated automatically.
- A compiler can warn about missing `case`s in `switch` statements over an enumeration.
- A **debugger** may also be able to print values of enumeration variables in symbolic form.

Definition syntax:

```
enum tag {
    name,
    name = val,
    ...
} variable...;
```

- the keyword `enum`
- an optional *enumeration* *tag*
- brace-enclosed list of *members*, sep. by **comma**, with optional assignment
- optional list of variables of the new type

- Declaring an enumeration is similar to `enum` and `struct`.

```
enum answer { no, yes };  /* definition */
enum answer x;  /* declaration */
```

```
/* shorthand */
enum { no, yes } x;
```

- An enumeration is a list of **constant integer values**.

```
enum answer { no, yes };

enum answer x;
int i;

x = no;
x = 42;      /* x is just an int */
i = yes;
no = 23;     /* invalid — not an lvalue! */
```

- If not assigned explicitly, the names are assigned consecutive integer constants, starting from 0.
- Enumeration continues from an explicit assignment.

```
enum months { JAN = 1, FEB, MAR, APR, MAY, JUN,
              JUL, AUG, SEP, OCT, NOV, DEC };
/* FEB is 2, MAR is 3 ... */
```